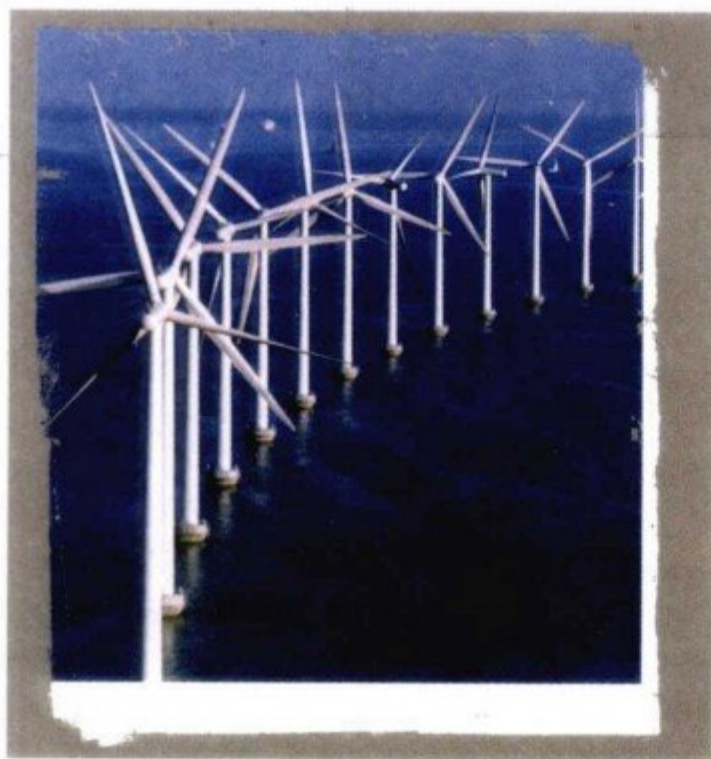




Pattern-Oriented Software Architecture **Volume 4**
A Pattern Language for Distributed Computing

面向模式的软件架构

分布式计算的模式语言



卷4

[德] Frank Buschmann
[英] Kevlin Henney 著
[美] Douglas C. Schmidt
肖鹏 陈立 译



人民邮电出版社
POSTS & TELECOM PRESS

Pattern-Oriented Software Architecture **Volume 4**

A Pattern Language for Distributed Computing

面向模式的软件架构 **卷4**

分布式计算的模式语言

迄今为止，人们提出的软件开发模式有不少是关于分布式计算的，但人们始终无法以完整的视角了解分布式计算中各种模式是如何协同工作、取长补短的。构建复杂的分布式系统似乎成为了永远也无法精通的一门手艺。本书的出版改变了这一切。

本书是经典的POSA系列的第4卷，介绍了一种模式设计语言，将分布式系统开发中的114个模式联系起来。书中首先介绍了一些分布式系统和模式语言的概念，然后通过一个仓库管理流程控制系统的例子，介绍如何使用模式语言设计分布式系统，最后介绍模式语言本身。

使用这一模式语言，人们可以有效地解决许多与分布式系统开发相关的技术问题，如

- ★ 对象交互
- ★ 接口与组件划分
- ★ 应用控制
- ★ 资源管理
- ★ 并发与同步

本书从实用角度展示了如何从现有的主要模式中整合出一门全面的模式语言，用于开发分布式计算中间件及应用程序。作为该领域在市场上唯一统揽全局的书，它将给读者带来醍醐灌顶的感觉！



WILEY

www.wiley.com

图灵网站: www.turingbook.com 热线: (010)51095186

反馈/投稿/推荐信箱: contact@turingbook.com

有奖勘误: debug@turingbook.com

分类建议 计算机/软件工程

人民邮电出版社网址: www.ptpress.com.cn



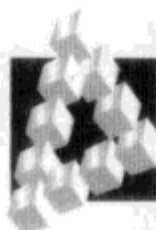
ISBN 978-7-115-22773-7



9 787115 227737 >

ISBN 978-7-115-22773-7

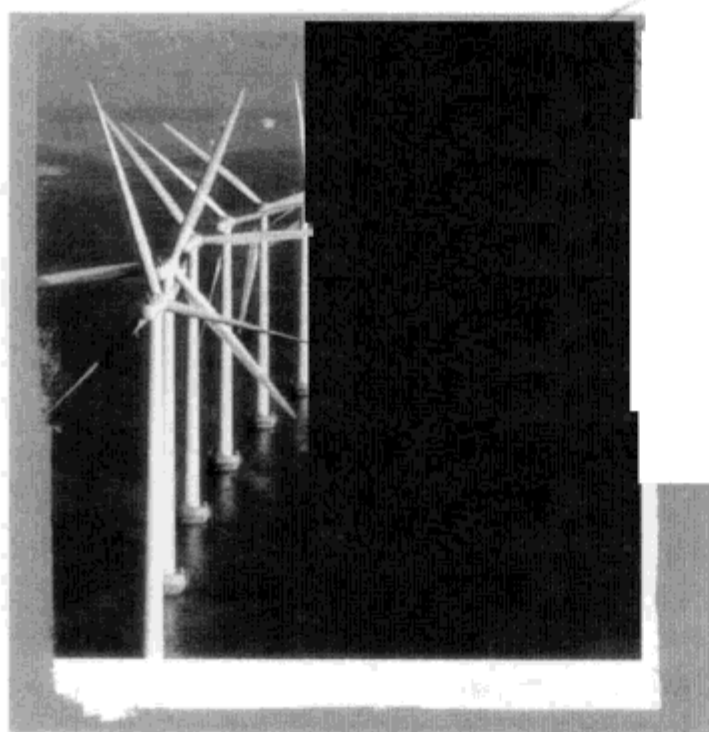
定价: 69.00元



Pattern-Oriented Software Architecture **Volume 4**
A Pattern Language for Distributed Computing

面向模式的软件架构

分布式计算的模式语言



卷4

Frank Buschmann

evlin Henney

Douglas C. Schmidt

肖鹏 陈立

著

译

人民邮电出版社
北京

图书在版编目 (C I P) 数据

面向模式的软件架构 卷4: 分布式计算的模式语言 / (德) 布施曼 (Buschmann, F.), (英) 亨尼 (Henney, K.), (美) 施密特 (Schmidt, D. C.) 著; 肖鹏, 陈立译. -- 北京: 人民邮电出版社, 2010. 6

(图灵程序设计丛书)

书名原文: Pattern-Oriented Software Architecture

ISBN 978-7-115-22773-7

I. ①面… II. ①布… ②亨… ③施… ④肖… ⑤陈… III. ①软件设计②分布式计算机系统 IV. ①TP311.5②TP338.8

中国版本图书馆CIP数据核字(2010)第071266号

内 容 提 要

本书关注分布式计算系统软件的设计和实现。书中首先介绍理解本书内容所需的核心的模式概念, 分布式计算的好处和挑战; 然后描述如何使用分布式计算模式语言, 设计真实世界中仓库管理流程控制系统; 最后重点讲述分布式计算模式语言, 该语言陈述了创建分布式系统相关的技术主题。

本书适用于软件架构师和开发人员。

图灵程序设计丛书

面向模式的软件架构 卷4: 分布式计算的模式语言

◆ 著 [德] Frank Buschmann [英] Kevlin Henney
[美] Douglas C. Schmidt

译 肖 鹏 陈 立

责任编辑 傅志红

执行编辑 贾利莹

◆ 人民邮电出版社出版发行 北京市崇文区夕照寺街14号

邮编 100061 电子函件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

中国铁道出版社印刷厂印刷

◆ 开本: 800×1000 1/16

印张: 23

字数: 538千字

2010年6月第1版

印数: 1-3 000册

2010年6月北京第1次印刷

著作权合同登记号 图字: 01-2008-0488号

ISBN 978-7-115-22773-7

定价: 69.00元

读者服务热线: (010)51095186 印装质量热线: (010)67129223

反盗版热线: (010)67171154

版 权 声 明

Original edition, entitled *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*, by Frank Buschmann, Kevlin Henney, Douglas C. Schmidt, ISBN 978-0-470-05902-9, published by John Wiley & Sons, Inc.

Copyright ©2007 by John Wiley & Sons, Inc., All rights reserved. This translation published under License.

Translation edition published by POSTS & TELECOM PRESS Copyright ©2010.

Copies of this book sold without a Wiley sticker on the cover are unauthorized and illegal.

本书简体中文版由 John Wiley & Sons, Inc. 授权人民邮电出版社独家出版。

本书封底贴有 John Wiley & Sons, Inc. 激光防伪标签，无标签者不得销售。

版权所有，侵权必究。



关于作者

Frank Buschmann

Frank Buschmann 是德国慕尼黑西门子技术公司的高级总工程师。他的研究领域包括对象技术、软件架构、产品线、模型驱动软件开发和模式。他在该领域著作甚多，其中最引人注目的便是POSA系列的前两卷[POSA1][POSA2]和最近的两卷：本书和[POSA5]。Frank在1992年至1996年曾是ANSI C++标准化委员会X3J16的成员，于1996年发起了首届EuroPLoP会议，与人合作汇编了数本模式方面的书籍[PLoPD3][SFHBS06]，现任Wiley软件设计模式丛书的主编。在西门子的开发工作中，Frank领导过几个大型工业软件项目的架构设计和实现，包括业务信息、工业自动化和电信系统。

工作之余，Frank的生活丰富多彩。他喜欢陪着妻子Martina和女儿Anna享受家庭的快乐；他喜欢骑着马儿Eddi外出遛弯，或者独自跑到慕尼黑啤酒花园打发时间；他喜欢在观看最爱的多特蒙德足球队的比赛时尽洒激情，也偶尔在慕尼黑歌剧院里的演出中尽情陶醉；临睡前他还喜欢来杯珍藏的苏格兰纯麦威士忌。

Kevlin Henney

Kevlin Henney住在英国布里斯托尔，是一名独立顾问。他主要从事自己感兴趣的领域的教授、辅导和实践，这些领域包括编程语言和技术、软件架构、模式和敏捷开发。他的客户既有全球化的公司，也有刚起步的小公司，所涉及的领域包括系统软件、电信、嵌入式系统、中间件开发、业务信息和金融。

Kevlin经常在各种软件会议上应邀演讲，同时也参与了多个会议的组织工作，包括EuroPLoP。他通过英国标准学会（BSI）和ISO参与到C++的标准中来，同时也参与了其他语言的标准化工作。Kevlin也因其写作受人关注，经常发表会议论文，主持各种出版物上定期的（和不定期的）专栏，比如C++ *Report*、C/C++ *Users Journal*、*Java Report*、*JavaSpektrum*、*Application Development Advisor*、*The Register*、*EXE*和*Overload*。

Kevlin的业余时间则是与妻子Carolyn还有两个孩子Stefan和Yannick度过的。他跟孩子摆摆积木，给孩子们修修玩具，看书，有时也啜几口啤酒或者来一杯葡萄酒。

Douglas C. Schmidt

Doug Schmidt是美国田纳西州纳什维尔市范德比尔特大学计算机科学教授，计算机科学与工程计划副主席。他的研究领域包括模式和模式语言、优化原理，还包括对于支持服务质量（QoS）的组件中间件相关技术的实证分析（*empirical analysis*）和支持分布式实时嵌入式系统的模型驱动的工程工具。

Doug是国际公认的软件开发方面的专家，尤其是模式、面向对象框架、实时中间件、建模工具和开源软件开发等方面。他已经在各种顶级技术杂志和会议上发表了300多篇论文；他与人合著了模式方面的著作[POSA2]和有关C++网络编程的[SH02][SH03]；同时他还与人合编过数本畅销的书，比如模式方面有[PLoPD1]，框架方面有[FJS99a][FJS99b]。除了学术研究，Doug还领导了ACE、TAO、CIAO和CoSMIC的开发，这些广泛应用的开源的中间件框架和模型驱动的工程工具包含了一整套丰富的可重用的组件，这些组件的实现大量使用了本书中介绍的模式。

在他闲暇之隙，Doug总是跟妻子Lori和儿子Bronson待在一起，他们一起练练举重、弹弹吉它、讨论世界历史和政治，或者开着他们的雪佛兰Corvette跑车四处兜风。



前言

模式运动已经进行了十多年，从追捧到棒杀再到慢慢接受，模式已经经历了这个常见的轮回。Frank、Doug和Kevlin一直参与其中，受到过赞美，也遭遇过嘲讽，重要的是他们从中收集了大量好的想法，并将其描绘出来。POSA系列图书被认为是模式相关文献中最为坚实的基础性著作之一，它的每一卷都在我的书架上占有一席之地。

POSA的前几卷属于传统的模式书籍，描绘了某些特定领域中使用的模式，其中大部分以前均未有书面记录。本书则不同。分布式计算是一个相当宽泛的主题，一卷图书哪怕只是容纳已知的模式也是远远不够的。实际上这些模式分布在很多书里面，包括POSA系列和一些别的书。本书的目的是要把它们聚在一起。所以，这里列出的模式可能比你平时看到的要多，当然其描述也要简洁得多。有些模式可能并不是主要关于分布式的，但是多少都会和分布式系统有些关系。因此，本书是以分布式系统为背景来介绍这些模式的用法，并加以总结。

本书并不是仅仅介绍每个独立的模式的——同时也介绍它们之间的关系。在任何一个系统中都会同时使用多个模式，然而，就拿我的体会来说，讲述其中的关系要比介绍单独的模式难得多。本书没有回避这个问题，书中给出了很多关于在分布式场合下联合使用多种模式的建议。

分布式往往是一个棘手的难题。事实上，经常有人引用我的所谓分布式对象设计第一定律的“名言”：“不要使用分布式对象。”我这样说是原因的——分布式使得软件设计更困难，所以我一直建议尽可能地避免采用分布式设计。然而无论我如何强烈地质疑分布式设计的范围，分布式毕竟是很多软件系统重要的组成部分。既然分布式这么难，分布式设计便更值得我们认真地研究一番——因此，本书也应该是每个程序员必备的图书了。

Martin Fowler^①

① Martin Fowler，软件开发方面著名作家和演讲者，ThoughtWorks公司首席科学家，《重构：改善既有代码的设计》作者。——译者注

关于本书

分布式计算正在将整个世界连接起来，创造一个公平的竞争环境[Fri06]。今天，无处不在的Web和电子商务更为分布式计算提供了更大的动力：人们需要连接并访问大量分散在世界各地的信息和服务。即时消息和聊天室在Internet上的流行则带来了另一方面的需求：与家人、朋友、同事和客户保持连接。其他推动分布式计算的动力包括提高性能、可伸缩性和容错能力，同时通过共享昂贵的硬件和外围设备来降低成本。

鉴于分布式计算在我们的职业和个人生活中的重要性，软件文献中的众多模式都把目光投向了这个主题[POSA1] [POSA2] [POSA3] [Lea99] [VSW02] [VKZ04] [HoWo03] [PLoPD1] [PLoP2] [PLoPD3] [PLoPD4] [PLoPD5]。遗憾的是，这些模式大多都是孤立地描述，引用的几个其他模式也大多出自同一本出版物。尽管每个独立的模式也有其用处，但是孤立的描述毕竟无法提供完整的视角来着重展示，在分布式计算系统中相关的模式是如何相互取长补短的。于是，构建复杂的分布式系统成了只有少数天才和大师们才能掌握的暗黑艺术^①。

为了展示更为完整的视角，本书描绘了一个单独的模式语言，它把多个与分布式计算有关的模式联系起来。在该语言中，每个模式或者直接与分布式计算有关，或者在分布式计算环境中扮演重要的支持角色。模式语言为分布式计算关键领域的最佳实践提供了指南，同时也提供了一个交流的工具。

目标读者

我们关注的是分布式计算系统软件的设计和实现。因此，本书的主要读者是专业的软件架构师或者是高阶的学生，他们参与开发分布式计算系统，设计新的应用或者改善和重构已有的应用。模式语言展示了一系列丰富的模式，目标是帮助架构师为分布式系统创建可持续的设计，并且帮助他们全面而专业地陈述其需求。

本书的第二类读者则是在工作中使用组件和通信中间件的开发人员。模式语言为开发人员提供了分布式系统设计的当前实践状态，以便他们可以更有效地使用中间件。可以从模式语言中获益的第三类人是项目和产品经理。这门语言使得产品经理对于他们所领导开发的系统的基本功能可以有更深入的理解，并且为他们和软件架构师以及开发人员沟通提供有用的词汇。

不过，我们并不打算让终端用户或者客户直接使用模式语言。正确地使用真实世界的隐喻可

① “暗黑艺术”，20世纪产生的一种艺术形态，其典型形式包括恐怖、抽象、神秘等。——译者注

能会使这些读者接受模式语言，但这需要换一种表达方式。此外，本书也不是分布式计算的全面指导手册。虽然我们讨论了这个主题的众多方面，并且包含了一个广泛的术语集，但是读者需要事先对分布式计算的核心概念和机制有所了解，比如死锁、事务处理、同步、调度和合意（consensus）。更多的和分布式计算相关的主题，比如网络协议和操作系统的设计，可以在参考文献中找到。

结构和内容

本书内容分为3个部分：概念、模式故事和模式语言。

第一部分“概念”介绍本书的背景：理解本书所需的核心的模式概念，分布式计算的好处和挑战的概述，支持分布式的技术的总结以及对模式语言的介绍。

第二部分“模式故事”描述如何使用分布式计算模式语言设计真实世界中仓库管理的流程控制系统。这个故事关注于该软件系统的3个领域：基线架构、通信中间件和仓库拓扑表现。

第三部分“模式语言”构成了本书的主体部分。它包括一门分布式计算模式语言，该语言陈述了与分布式系统结构相关的技术主题：

- 给出初始的软件基线架构
- 理解通信中间件
- 事件分离和分发
- 接口划分
- 组件划分
- 应用控制
- 并发
- 同步
- 对象间的交互
- 适配与扩展
- 模态行为
- 资源管理
- 数据库访问

每一章介绍了该章的主题，总结了主要的挑战，然后展示了帮助应对这些挑战的一系列模式。我们的分布式计算模式语言总共包含了114种模式，并且连接到其他文献中介绍的150多种模式。这可以称得上是目前为止较为巨大的，甚至是最大的已经文档化的软件模式语言了。

虽然分布式计算是该语言的关注点，但是其中很多内容具有更广泛的适用性。例如，大多数的应用程序必须具有某种形式的适应性和可扩展性，而且每个软件系统都需要良好设计的接口和组件。对于相应的技术领域，模式语言可以作为现代软件开发中最佳实践的总体指南，而不是仅限于分布式计算领域。

本书的结尾包含了对分布式计算模式语言的简单回顾、常用的术语词汇表、一个庞大的该领域参考文献列表。

毫无疑问，我们不可能覆盖到分布式系统所有的属性和模式，更何况随着时间的推移，在模式语言的实际应用和扩展实践中还会有更多的模式涌现出来。如果你有任何关于改进本书的风格和内容的评价、建设性的批评或建议，请通过电子邮件发送给siemens-patterns@cs.uiuc.edu。在模式主页<http://hillside.net/patterns>有一份注册指南。该链接还提供了一个关于模式的方方面面的重要信息源，比如已经出版的和即将出版的图书、模式会议、模式论文等。

致谢

我们很高兴可以在此感谢帮助我们完成此书的人们，他们有的是通过与我们分享他们的知识，有的则审阅了早期的各个章节的草稿。

审阅奖应该发给Michael Kircher，他认真审阅了我们的材料，包括其正确性、完整性、一致性和总体质量。Michael的反馈极大地提高了本书的质量。

另外，我们在3次EuroPLoP模式会议上展示了该语言的几个部分，同时也给分布式和模式方面的几个专家做了展示。Ademar Aguiar、Steve Berczuk、Alan O'Callaghan、Ekatarina Chtcherbina、Jens Coldewey、Richard Gabriel、Ian Graham、Prashant Jain、Nora Koch、Doug Lea、Klaus Marquardt、Andrey Nechypurenko、Kristian Sørensen、James Siddle、Michael Stal、Steve Vinoski、Markus Völter、Oliver Vogel和Uwe Zdun为我们提供了大量的反馈，该语言的很多大小修订均是由此而来。

非常感谢Mai Skou Nielsen，她在丹麦奥尔胡斯市的JAOO 2006会议上为Kevlin和Frank照了一张照片。Anton Brøgger帮助查明了我们在第12章所使用的照片的情况。Publicis Kommunikation-sagentur GmbH和Lutz Buschmann允许我们在本书中使用他们的一些照片。

特别感谢Lothar Borrmann和Reinhold Achatz给予的行政上的支持和德国慕尼黑西门子技术股份公司软件工程实验室的支持。

尤其感谢我们的编辑Sally Tickner，我们以前的编辑Gaynor Redvers-Mutton，以及John Wiley & Sons的其他人员，没有他们便不会有本书的出版。尽管我们的日常工作已经非常繁忙，Gaynor还是说服我们撰写了本卷POSA。接下来，Sally在我们撰写本书的数年里表现出极大的耐心。特别感谢我们的文字编辑，WordMongers公司的Steve Rickaby，他令本书增色不少。本书是由Steve负责的POSA系列的第4卷，我们希望后面的几卷还能跟他合作。

最后，还要感谢我们的家人，感谢他们在本书撰写期间给予的耐心和支持！



读者指南

你能做到，只是不要想一步登天。

——Oprah Winfrey^①

本书的编写结构允许你用任何自己喜欢的方式阅读。最简单的方式就是从头读到尾。如果你自己知道想要了解哪一块，也可以自由地选择自己的阅读路径。这里，我们给出了一些线索，希望能帮助你找到自己关注的内容，并确定阅读的顺序。

关于模式和模式语言

本书为读者呈现了一种分布式计算模式语言，它包含了一系列相关的模式。这些模式定义了应该按照什么样的流程，系统地解决开发分布式系统软件过程中出现的问题。我们编写本书的目的是帮助你在日常的软件开发活动中应用这些模式来创建可工作、可支撑（sustainable）的分布系统软件架构。我们假定你对于模式和模式语言的概念已经有一定的了解，所以本书并不是专门全面介绍这两个概念的教程。

如果本书是你第一次接触到模式，我们建议你首先阅读《面向模式的软件体系结构（第一卷）：模式系统》[POSA1]和《设计模式：可复用面向对象软件的基础》[GoF95]中关于模式的介绍。这两本书介绍了软件架构和设计中有模式的基本概念和术语。如果你已经熟悉了模式的概念，而不了解模式语言的概念，我们建议你阅读第1章，以及James O. Coplien写的*Software Patterns*的白皮书[Cope96]，从中你可以了解到模式语言的概念，这些内容能够保证你从本书的分布式计算模式语言中有所收获。以上两部分也简要地介绍了模式概念方面的高级内容，其思想要比[POSA1]和[GoF95]中的深一些。

关于分布式计算

本书假定你对分布式计算的关键概念和机制有一定的了解。第2章简要地描述了分布式计算的优点和挑战，并总结了支持分布式的技术，但并没有详细地讨论分布式计算和分布式系统。这一章的目的只是从总体上介绍一下本书的主题：要获得分布式计算的好处，你应该踏踏实实地根据我们的模式语言的指导去应对相关的挑战。

^① 美国著名脱口秀主持人。——译者注

如果你需要更多关于分布式计算的背景知识，我们建议你阅读Andrew S. Tanenbaum和Maarten van Steen的*Distributed Systems: Principles and Paradigms* [TaSte02]和Ken Birman的*Reliable Distributed Systems* [Bir05]。

关于分布式计算模式语言

在开始阅读模式语言中的全部或者某几个模式之前，我们建议你先读第3章模式语言。这一章从总体上介绍了我们的模式语言，其中主要包括如下内容。

- 意图、范围和读者。
- 模式语言的通用结构，模式语言中与分布式计算相关的主题和要解决的挑战，以及该模式语言包含的具体模式。
- 介绍和阐述模式语言中的模式时，模式所使用的形式和标记法。

这一章也是模式语言的一个通用地图，有了这张地图，当你读到某一个或一组特定的模式时，就知道自己位于什么地方了。这个地图的目的就是帮助你在阅读某个模式的细节时，避免出现只见树木不见森林的尴尬。

模式语言实践

本书的第二部分，给出了一个具体的例子，阐明在实践中如何应用分布式计算模式语言为仓库管理流程控制系统创建架构。通过创建一个真实世界的系统，展示了分布式计算模式语言在构建高质量软件系统中可以为架构师和开发人员提供怎样的信息。如果你擅长通过例子学习，我们建议你在深入阅读模式语言之前先把这个例子看一遍，虽然先学习模式语言再回来看这个例子并不影响例子所表达的信息。这个例子演示了我们的模式语言如何帮助你创建和理解如下功能。

- 分布式系统的基线架构。基线架构可以高效地划分系统的功能和基础设施职责，并保证系统满足服务质量要求。
- 通信中间件。中间件使得分布式系统的组件之间能够高效地、健壮地、可移植地交互。
- 分布式系统中具体组件的详细设计。这些设计应该能够支持分配给自己的职责，满足各自的需求。

虽然这个例子本身就是完备的，但要达到融会贯通的程度还是需要阅读相应的章节，直到找到所描述问题的最基础的解决方案陈述。这时，如果你不熟悉某个模式，我们建议你阅读第三部分中相关模式的概要。如果你熟悉这个模式，可以继续阅读这个例子，看看这个模式是如何在仓库管理系统中应用的，备选的模式还有哪些，为什么没有选择备选的模式。

模式语言细节

本书的第三部分模式语言详细地介绍了分布式计算模式语言，包括与构建分布式系统相关的关键技术主题。我们建议你采用下面的方式阅读这一部分内容。

- 从头至尾。在这一部分有关语言的技术主题以及与之相关的模式，大体上是按照我们构

建分布式系统时的相关性和应用的顺序编写的。

- 按主题阅读。如果你对某个技术主题特别感兴趣，比如组件划分，你可以只阅读相应的章节。每一章的开头列举并讨论了与相应的技术主题有关的挑战，介绍了帮助我们应对这些挑战的模式，对比了它们的相同点和不同点。之后对每个模式的扼要介绍，也是每一章的重点。
- 按模式阅读。最后，如果你只是对某个模式感兴趣，你可以根据第3章中列出的模式，找到该模式在语言中的位置，直接阅读相关的内容。

模式概要并未介绍模式的实现细节，比如如何使用某个编程语言或者在某个特定的中间件平台上实现这个模式。我们在每个模式的概要中展示并讨论了该模式所解决的问题、模式背后的驱动力、它所包含的关键解决方案，以及应用该模式会带来结果。同时，我们也说明了要实现这个模式可能用到的语言中的其他模式，以及该模式可以应用于哪些其他模式的实现。整本书中，当我们提及语言中某个模式的时候，我们会在其后用括号注明其所在页数。如果你对某个模式的实现细节感兴趣，我们建议你查阅我们引用的原始出处。



目 录

第一部分 概 念

第 1 章 模式与模式语言	2
1.1 模式	2
1.2 模式内幕	3
1.2.1 问题的环境	3
1.2.2 驱动因素: 所有模式的核心	4
1.2.3 解决方案与结果	4
1.2.4 模式命名	4
1.2.5 模式表现形式概述	5
1.3 模式的关系	5
1.3.1 模式的互补	5
1.3.2 模式的组合	6
1.3.3 模式故事	6
1.3.4 模式序列	7
1.4 模式语言	7
1.4.1 从模式序列到模式语言	7
1.4.2 展现和使用模式语言	7
1.5 模式的连接	8
第 2 章 分布式系统	9
2.1 分布式的优点	9
2.2 分布式的挑战	11
2.3 用以支持分布式的技术	12
2.3.1 分布式对象计算中间件	13
2.3.2 组件中间件	14
2.3.3 发布/订阅中间件和面向消息的中间件	15
2.3.4 面向服务架构和 Web 服务	16
2.4 中间件技术的局限性	17
第 3 章 模式语言	18
3.1 意图、范畴和对象	18

3.2 起源	18
3.3 结构和内容	19
3.4 模式的表现	24
3.5 实际应用	26

第二部分 模式故事

第 4 章 仓库管理流程控制	33
4.1 系统范畴	33
4.2 仓库管理流程控制	34
第 5 章 基线架构	37
5.1 架构环境	37
5.2 划分大泥球	38
5.3 层次分解	38
5.4 访问领域对象功能	40
5.5 网络桥接	41
5.6 分离用户界面	43
5.7 功能分布	45
5.8 支持并发的领域对象访问	47
5.9 获得可扩展的并发性	48
5.10 将面向对象与关系型数据库连接起来	49
5.11 领域对象的运行时配置	50
5.12 基线架构总结	51
第 6 章 通信中间件	54
6.1 分布式系统的中间件架构	54
6.2 对中间件的内部设计进行结构化	57
6.3 封装底层系统机制	58
6.4 分离 ORB 核心事件	59
6.5 ORB 连接管理	61
6.6 提高 ORB 的可伸缩性	63
6.7 实现同步请求队列	65

6.8 可互换的内部 ORB 机制.....	66	10.6 Publisher-Subscriber**	135
6.9 管理 ORB 策略	68	10.7 Broker**	137
6.10 ORB 动态配置	69	10.8 Client Proxy**	139
6.11 通信中间件总结	71	10.9 Requestor**	140
第 7 章 仓库拓扑	74	10.10 Invoker**	142
7.1 仓库拓扑基线	74	10.11 Client Request Handler**	143
7.2 表现层次化的存储结构	74	10.12 Server Request Handler**	144
7.3 存储结构导航	77	第 11 章 事件分离和分发	147
7.4 存储属性建模	78	11.1 Reactor**	150
7.5 不同的存储单元行为	79	11.2 Proactor*	152
7.6 实现全局功能	81	11.3 Acceptor-Connector**	154
7.7 遍历仓库拓扑	81	11.4 Asynchronous Completion Token**	155
7.8 支持控制流扩展	83	第 12 章 接口划分	157
7.9 连接数据库	84	12.1 Explicit Interface**	163
7.10 维护内存中的存储单元数据	85	12.2 Extension Interface**	165
7.11 配置仓库拓扑	86	12.3 Introspective Interface**	166
7.12 细述显式接口	88	12.4 Dynamic Invocation Interface*	167
7.13 仓库拓扑总结	89	12.5 Proxy**	169
第 8 章 模式故事背后的故事	91	12.6 Business Delegate**	170
第三部分 模式语言		12.7 Facade**	171
第 9 章 从混沌到结构	97	12.8 Combined Method**	172
9.1 Domain Model**	106	12.9 Iterator**	173
9.2 Layers**	108	12.10 Enumeration Method**	174
9.3 Model-View-Controller**	109	12.11 Batch Method**	175
9.4 Presentation-Abstraction-Control	111	第 13 章 组件划分	177
9.5 Microkernel**	113	13.1 Encapsulated Implementation**	181
9.6 Reflection*	114	13.2 Whole-Part**	183
9.7 Pipes and Filters**	116	13.3 Composite**	185
9.8 Shared Repository**	117	13.4 Master-Slave**	186
9.9 Blackboard	119	13.5 Half-Object plus Protocol**	188
9.10 Domain Object**	121	13.6 Replicated Component Group**	189
第 10 章 分布式基础设施	123	第 14 章 应用控制	191
10.1 Messaging**	129	14.1 Page Controller**	196
10.2 Message Channel**	130	14.2 Front Controller**	197
10.3 Message Endpoint**	132	14.3 Application Controller**	198
10.4 Message Translator**	133	14.4 Command Processor**	199
10.5 Message Router**	134	14.5 Template View**	200

14.6 Transform View**	201	18.12 Wrapper Facade**	269
14.7 Firewall Proxy**	202	18.13 Declarative Component Configuration*	270
14.8 Authorization**	204	第 19 章 模态行为	272
第 15 章 并发	206	19.1 Objects for States*	274
15.1 Half-Sync/Half-Async**	209	19.2 Methods for States*	275
15.2 Leader/Followers**	211	19.3 Collections for States*	276
15.3 Active Object**	212	第 20 章 资源管理	278
15.4 Monitor Object**	214	20.1 Container*	288
第 16 章 同步	216	20.2 Component Configurator*	289
16.1 Guarded Suspension**	221	20.3 Object Manager**	291
16.2 Future**	223	20.4 Lookup**	292
16.3 Thread-Safe Interface*	224	20.5 Virtual Proxy**	294
16.4 Double-Checked Locking	225	20.6 Lifecycle Callback**	295
16.5 Strategized Locking**	226	20.7 Task Coordinator*	296
16.6 Scoped Locking**	227	20.8 Resource Pool**	298
16.7 Thread-Specific Storage	228	20.9 Resource Cache**	299
16.8 Copied Value**	230	20.10 Lazy Acquisition**	300
16.9 Immutable Value**	231	20.11 Eager Acquisition**	301
第 17 章 对象间的交互	233	20.12 Partial Acquisition*	303
17.1 Observer**	237	20.13 Activator**	304
17.2 Double Dispatch **	238	20.14 Evictor**	305
17.3 Mediator*	239	20.15 Leasing**	306
17.4 Command**	240	20.16 Automated Garbage Collection**	307
17.5 Memento**	242	20.17 Counting Handles**	309
17.6 Context Object**	243	20.18 Abstract Factory**	311
17.7 Data Transfer Object**	244	20.19 Builder*	312
17.8 Message**	245	20.20 Factory Method**	313
第 18 章 适配与扩展	247	20.21 Disposal Method**	314
18.1 Bridge**	255	第 21 章 数据库访问	316
18.2 Object Adapter**	256	21.1 Database Access Layer**	318
18.3 Chain of Responsibility*	257	21.2 Data Mapper**	320
18.4 Interpreter	258	21.3 Row Data Gateway**	321
18.5 Interceptor**	260	21.4 Table Data Gateway **	323
18.6 Visitor**	261	21.5 Active Record	324
18.7 Decorator	262	第 22 章 最后的思考	326
18.8 Execute-Around Object**	264	术语表	327
18.9 Template Method*	265	参考书目	340
18.10 Strategy**	266		
18.11 Null Object**	267		

Part 1

第一部分

概 念

语言之城，人人皆有一砖一石之贡献。

——拉尔夫·瓦尔多·爱默生^①

本书第一部分为分布式计算模式语言提供了背景。我们给出了模式和模式语言的概念、分布式计算的优点和挑战，并简单介绍了模式语言本身的概况。

本书关注的是模式和分布式计算模式语言。为了更好地理解这些模式和语言，并能在构建分布式系统产品的过程中成功应用它们，则了解模式和分布式计算相关的概念以及已有的分布式技术，不仅有益，而且必要。要在项目开发中有效地使用模式语言，你需要理解其范围、结构、内容和表现。

本书的第一部分总体描述了这些概念，同时也简单介绍了分布式计算模式语言。

第1章模式与模式语言介绍了模式和模式语言概念的各个方面，有助于我们理解分布式计算模式语言。我们给出了模式的基本概念，讨论了这些概念的核心属性，展示了这些模式是如何连接在一起组成模式语言的，以及模式所组成的网络如何相互配合、系统地解决软件开发中相互交织的各种问题。

第2章分布式系统简要地介绍了构建分布式系统的主要优点和挑战，指出了每一代分布式技术分别用于应对哪些挑战和如何应对这些挑战，以及还有哪些问题尚未解决，必须由分布式系统中的应用架构来解决。

第3章模式语言介绍了分布式计算模式语言。我们说明了语言的意图和范围，定义了它的适用性。我们还简单地介绍了所覆盖的13个问题领域，列出了114种模式，展示了语言的具体结构和范围。在本部分，我们还给出了语言的具体展现形式，比如模式的格式和用到的标记法，以及在产品开发生项目中如何支持应用。

这3章为全书设定了背景，我们在第二部分模式故事中给出了一个关于仓库管理流程控制系统的模式案例，在第三部分模式语言中详细介绍模式语言。

^① 爱默生（1803—1882），美国散文作家、思想家、诗人。——译者注

没有清晰深邃的思想则无从凸现语言的光彩，没有闪光的语言亦无法展现思想之美。

——马尔库斯·图留斯·西塞罗（公元前106—前43），
古罗马政治家、演说家和哲学家

本章首先简要地介绍了模式的概况，包括其历史和一系列基本概念。接下来，详细介绍了模式的组成结构、所提供的解决方案及其产生的原因。这里，我们还会探讨常见的不同模式之间的关系。最后，通过对模式语言的讨论，我们将总结出什么是模式语言以及如何表达并使用它们。

1.1 模式

从设计角度来说，软件常被认为由以下部分组成：函数、源文件、模块、对象、方法、类、包、库、组件、服务以及子系统等。它们从不同的角度和层面反映了开发人员的直接工作，并且都只关注于软件的局部，而忽略了这些元素之间更广泛的联系以及如此设计的原因。相比之下，模式则是一种补充，它可以用来描述和改良软件设计，发掘并命名那些已经验证的通用技术。模式不仅解释了什么是设计，它更强调了为什么设计，应该在什么时候设计以及怎样设计。

每个模式都记录了一个在特定环境下不断重复发生的问题以及相应的解决方案。然而，它并不局限于此。除了问题及解决方案本身之外，模式还包括了将问题和解决方案衔接起来的基本原理。其中“问题”分析了相互冲突的因素，并阐明了问题之所以成为问题的原因，而“解决方案”则详细描述了该方案的组成结构及其应用效果——优劣兼顾。

模式的可重复性是很重要的——这也是模式（pattern）得名的原因——这样才能一次又一次地依赖相同的经验支持而不需要重复劳动。发现在不同应用中设计的共性使得开发人员可以利用已经掌握的知识，以便在一个陌生的应用中运用熟悉的技术。当然，在很多情况下这可能仅仅被称为“经验”。模式超越个人经验之处在于：模式被正式命名并记录，更有利于提炼、交流和分享架构层次上的知识。

从建筑架构到软件架构

尽管设计模式现在很流行并普遍应用在软件开发领域中，但它并非起源于虚拟的软件世界，而是起源于现实的建筑世界。在20世纪60年代和70年代，著名的建筑师克里斯托弗·亚历山大和

他的同事们提出了模式的概念，以记录建筑中的架构组织和设计灵感[Ale79][AIS77]。他们尤其希望集中关注那些被实践证明的“完整”模式——可以改善居住环境、提高人们生活质量的模式。架构模式的提出也是对当代建筑中很多流行但不成功的潮流和实践的反思。

他们把所记录的模式组织成一个统一的集合，通过模式语言（pattern language）表达不同层次的抽象——城市、社区、家庭，并且将不同模式的运用有机地联系起来。这个方法首先由Kent Beck和Ward Cunningham[BeCu87]应用到一些人机交互相关的设计中，创建了几种用户界面设计模式，其中包含了大量关于面向对象编程模式语言的建议。相关的思想在随后几年里更多地出现，比如Jim Coplien关于C++惯用法和编程风格的书[Cope92]，Erich Gamma的框架设计论文[Gam92]以及Bruce Andersen的*Handbook of Software Architecture*。随着这股风潮的深入，Gang-of-Four^①的标志性著作《设计模式》[GoF95]出版了。

随着公众对软件设计模式的兴趣越来越高，一个专门的社区出现了，它专注于收集、记录模式，并通过一些领军人物和作者的讨论研究加以改进。许多模式被公开在网络、杂志以及随后出版的书籍中。其中之一便是*Pattern Language of Program Design*系列[PLoPD1][PLoPD2][PLoPD3][PLoPD4][PLoPD5]，它总结提炼了世界各地PLoP（编程模式语言）会议的成果。另一个就是*Pattern-Oriented Software Architecture*系列。第1卷《模式系统》[POSA1]是对《设计模式》的补充，它的一些模式是构建在Gang-of-Four的设计模式基础上的，或者是对它们的扩展，而且更明确关注不同扩展层次的软件架构，特别是大型系统。第2卷《用于并发和网络化对象的模式》[POSA2]和第3卷《用于资源管理的模式》[POSA3]则详细讨论了软件架构的特定方面——主要是并发和网络应用，而第4卷*A Pattern Language for Distributed Computing*正是你手中的这本。

1.2 模式内幕

设想如下场景。你打开壁橱——空的，然后看看冰箱——空的，只有灯是亮的。你知道该购物了，你需要牛奶、果汁、咖啡、比萨、水果等许多主食。你去了超市，进去，找到牛奶，拿起几瓶，结账，回家将其放入冰箱。然后又回到超市，进去，找到果汁，拿起几瓶，结账，再回家将其放入冰箱。不断重复直到你买齐所有需要的东西。

购物也许可以这样，但这并非有效的方式，即使超市近在咫尺，我们有更明智的方式有效利用时间。小的调整和优化不会带来性能的显著提升，比如出门的时候不锁门、不关冰箱门、使用现金而不是信用卡等，我们需要一种截然不同的购物方式。

方案二：列一个购物清单，去超市，进去取一个购物车，找到清单上的所有物品并放入购物车（可能会少一些缺货的物品，多一些一时冲动所购买的物品），结账回家，然后拆包并把所有东西放入冰箱。

1.2.1 问题的环境

零售店乍一看可能和软件开发毫无关系，但是我们刚才描述的例子在本质上是一个分布式编

^① “Gang-of-Four”指的是Erich Gamma、Richard Helm、Ralph Johnson和John Vlissides，他们以及他们的著作皆声名远扬。John Vlissides先生于2005年11月24日在家中去世。——译者注

程问题及其解决方案。这个编程任务是迭代（iteration）问题：遍历一个集合以完成一个特定的目标。当完成这个迭代任务的代码和被访问的集合对象在同一进程时，其访问开销几乎可以小到忽略不计。然而如果情况发生改变，那么这种假设以及相应的解决方案也就不再正确了。分布式系统为任何存取访问引入了巨大的开销——如果没有考虑到这一点就很可能做出低效的、僵化的、不合适的设计。对于模式以及设计来说也是如此：背景往往会导致特定的问题，任何解决方案都必须考虑到这一点。

1.2.2 驱动因素：所有模式的核心

背景设定了一个问题的场景，“驱动因素”则代表了问题的各个方面。“驱动因素”决定了一个有效的解决方案所必须考虑的内容。对于任何重要的设计决定来说，不可避免地会导致相应“驱动因素”之间的冲突。

回到分布式计算环境和遍历集合的问题，我们面对着一系列必须考虑的问题。网络上通信（或者说从家去超市）的时间开销很大，一个高效的解决方案不能对此视而不见。另一方面，由于值（或者食物）必须被封送和解封送（打包和拆包），所以还要考虑通信中涉及的时间和空间开销。我们还必须考虑到方便性：拿一盒牛奶，结账并放回冰箱，所需要的步骤肯定没有推一个购物车以及打包和拆包那么麻烦。细粒度的迭代通常更为程序员所熟悉，并得到库和语言更好的支持。最后，我们还必须考虑部分失败（或部分成功）的情况，比如“networks”可能会被错误地接收为“notworks”。

1.2.3 解决方案与结果

一个有效的解决方案不仅要平衡引起问题的“驱动因素”之间的冲突，还应该表达清晰。回到我们刚才所描述的问题，我们的方案包括准备和发送批量的请求（购物清单）至目的地，采用某种适当的协议（开车、步行、骑车等）来一次性完成所有的任务（用购物车在超市购物），并把结果封送传回调用方，解封送（unmarshal）到本地地址空间（结账，打包回家），然后在本地遍历所有数据（拆包并整理好物品）。

任何设计决定都不会是完美的，每个决定都会产生相应的效果，有些是正面的，有些则是负面的。因此一种模式的有效应用应当能带来显著的效益和相对微弱的负面效应，有时候负面效应甚至不曾显现出来。

还以我们刚才讨论的情形为例，可能主要的关注点是通信和计算之间的比率，因此需要一种只需要花费少量的通信成本（一次输入，一次输出）就可以完成全部任务（访问多个数据）的方案，并确保部分的失败不会导致状态的不完全——要么传回所有数据，要么一个也不传送。然而天下没有免费的午餐。与一次遍历不同，这个方案有多次遍历：一次调用前的本地遍历（准备购物清单），一次远程遍历完成计算并记录每一项的结果（在超市购物），一次本地遍历处理结果（拆包并整理货物）。这也是分布式系统的惯用设计风格，有别于大多数本地程序的实现。

1.2.4 模式命名

每个设计模式都应该有一个名称，这样可以丰富软件设计的词汇。模式名能在我们的对话中

提高语言的抽象层次，使我们不必每次都重复描述该模式的重要元素。

有效的模式名应当能明确指出解决方案的关键因素，通常都是些名词。这里所举的例子中的模式名为Batch Method (175)，这个名字可以帮助我们将它和其他适用于不同场合或解决不同问题的迭代模式区分开，比如Iterator (173)——购物例子中采用的第一种方案，以及Enumeration Method (174)——和Iterator正好相反，它将集合的循环封装起来，针对集合中的每个元素执行一小段代码。

1.2.5 模式表现形式概述

模式的名称大都基于某一种特定实现之上，但其在实践中的运用并不一定完全相同，只要具备了某模式的一些关键特征，我们就可以认为遵循了该模式。然而，为了软件架构师和开发人员更普遍广泛地使用设计模式，我们通常需要一个更具体的描述方式。前面我们已经详细描述了模式的结构，如模式的背景和驱动因素，那么我们怎样才能通过书面语言中去展现它呢？

对于这个问题有不只一种答案。在实际应用中，针对不同的读者或阅读习惯，模式有很多种表现形式，各有其不同的侧重方面。例如，有的模式对某一特定编程语言来说是通用的，对它进行全面介绍的文档经常在较高层次上描述技术细节，并带有示例代码，最好还用一或多个例子来辅助描述。对于很长的模式，可以把它们分成几段并给它们分别命名，以便于读者和潜在的用户更好地理解。相反，对于把一个模式和许多其他模式放在一起作为基本开发过程的指南的情形，用大篇幅的细节描述和代码来表达可能不太适合。这种情况下，用一个更加简洁的形式可能更好一些。

无论模式采用什么样的形式，都应当清楚地描述其核心问题和解决方案，强调驱动因素及其效果，并包含对目标受众来说尽可能多的有用的结构、图形和技术细节。

1.3 模式的关系

模式可以被单独使用并带来某种程度上的成功，它可以反映问题的讨论、对特定问题点的解决方案以及局部的设计思想。然而更多时候，一种模式会和其他模式结合起来运用，任何已知的应用程序或库都会使用大量的设计模式。

我们可以按照不同的标准对模式进行分类——面向对象的框架模式、企业级计算模式、安全模式、特定编程语言模式等。在这些情况下最重要的事情可能是怎样将这些模式联系起来。一个按字母顺序的模式列表可能有助于根据名称来查找特定模式，但并不能描述它们之间的关系。

软件架构是包含许多不同决定的连锁网络，每个决定都与矛盾、建议以及其他决定有关。因此，为了使一个架构概念具有实际意义，模式之间在表达和变化上会不可避免地产生合作或竞争关系。

1.3.1 模式的互补

在实践中设计往往很容易墨守成规，对某类问题总是运用某一特定模式。虽然它经常会成功，但很多时候，习惯性地选择某个模式可能不是最高效的，反而是最低效的。就像Emile-Auguste

Chartier说的：“如果你只有一个设计观点，那么没有什么比这个观点更危险的了。”

设计词汇与其他词汇一样，有效表达部分是建立在词汇的广度上的，特别是同义词，每个都会在内涵上有一些细微的差别。两个或两个以上的模式可能看起来解决了同一个或相似的问题——比如Iterator和Batch Method。决定采用哪一个需要很好地理解问题的背景、目标、驱动因素以及折衷方案。从这一点来说，我们可以认为模式之间是互补的，因为它们联合提供了基于某个设计决策的不同方案。

模式也可以互相协作，一个模式可以提供另一个模式所缺少的部分，其目标是使得设计结果更加平衡和完整。从这一点上我们也可以认为模式是互补的，因为它们共同存在并相互促进。此外，许多被认为可以相互替代的存在竞争关系的模式，比如Iterator和Batch Method，也可以在彼此协作中互为补充。

再考虑一下我们的分布式迭代问题。通过在每次循环中访问一个元素，Iterator本身提供了一种对远程集合访问来说细粒度的、效率低的方案。与之相反，Batch Method用重复访问数据代替了重复访问网络——传送和接收集合，这在大多数情况下能很好地工作，但是对于很大的集合来说，或者对于需要及时响应的客户端来说，用于封送、发出、接收、解封送的时间会使得客户端长时间没有响应。一个可能的备选方案是将二者结合：使用Iterator的基本思想来遍历，但不是每次处理一个元素，而是用Batch Method每次处理若干个。

1.3.2 模式的组合

模式的组合是指重复出现的某些模式的集合，这些模式经常一起出现，以至于可以认为这是对某个特定可重复性问题的一套解决方案。比如在分布式计算中，Iterator和Batch Method的结合就是这样一个例子，它常被称为Batch Iterator或Chunky Iterator。

模式的组合又被称为复合模式（compound patterns）。它最初被称为组合模式（composite patterns），然而，在平时的交流中，它很容易和GOF的《设计模式》中众所周知的Composite（185）模式”相混淆。

实际上，大多数模式在某种程度上，或者从某种角度来看都是复合的，所以从本质上来说，复合模式的概念和设计的粒度有关。

1.3.3 模式故事

一个系统的开发可以被认为是一整篇的描述文章，其中一个又一个的设计问题被提出，被解答，并按照特定的缘由来组织结构等。我们可以把许多设计的出现和萃取看作是相应模式的不断应用。由一段描述引入设计——一个模式故事——构造一个又一个设计模式以解决设计中的问题，或者是解决上一个模式遗留下来的问题。

正如其他的故事情节一样，它们抓住所发生问题的精髓，尽管并不一定就是事实。在描述设计及其演化中逐步提供比实际更多的内容。在实际中很少会有设计思想能正好对应到某一场景中，因此随着时间的推移在对设计复盘时适度地反思、修改以及重新组织是很重要的。

这些故事，可以是已完成的系统，也可以是对将要构建系统的预见或者仅提供怎样构造系

统的假设。它们可能是完全从实际开发过程中提取出来再经过理想化处理的，进而作为在架构和设计论证中的指导。它们还可能被用作故事板方法（storyboarding technique），以便展望和探索将来的设计决策和系统方向。它们甚至可以是纯理论的和理想化的内容，用来传授或探索设计思想，而并非实际系统。

1.3.4 模式序列

模式序列与模式故事之间的关系，就如同独立的模式与描述并促使这些模式产生的例子之间的关系一样：它们归纳了模式的进程并确立了一个设计的方法，而不一定要成为一个特定的设计。从这点上来说，一个给定的模式序列可以被认为是高度抽象的开发过程，前一个模式成为后续设计模式场景的一部分。

例如，Batch Iterator是对Iterator和Batch Method的应用，因此也可以被认为是一个（非常）短的模式序列，其中Iterator用于提供遍历的概念，而Batch Method用于定义每次访问的形式。

1.4 模式语言

模式是设计中的词汇，而模式语言则在某种程度上相当于语法和风格。通过使用模式，模式语言可以在以下方面提供指导：怎样创建特定类型的系统，怎样实现特定的类，怎样去满足某个特定的横切交叉点需求，以及怎样去设计某类产品，等等。不管我们是想构建一个分布式的仓库管理系统，还是想使用C++编写异常安全（exception-safe）的代码，抑或打算开发一个基于Java的互联网应用，如果有这些领域相关的经验，那么我们很可能会把这些经验总结成模式，并组织成模式语言。

1.4.1 从模式序列到模式语言

与模式故事的具体和直接不同，模式序列更加抽象。但其本质上仍然是线性的，设计人员会从模式序列的反馈信息中得知如何应用下一个模式，或者得知是否需要重新思考前一个模式的应用。模式语言则要抽象得多，而且彼此之间的联系更紧密。

模式语言定义了一系列互相联系的模式，一个模式——可选或者必须——依赖于另一个模式，形成一个树或有向图，从而以一种特定的方式详细描述设计，响应特定的驱动因素，并根据情况选择适当的解决方法。在1.3节中所描述的各种关系都能够在模式语言中找到对应的形式。例如，模式序列通过模式语言定义了一条路径，包括部分或所有相关的模式，一个模式故事对应其路径上的一条路线。

1.4.2 展现和使用模式语言

模式语言包含模式序列，而怎样处理模式路径上的反馈等相关知识也应当被认为在模式语言的责任范畴之内。一个给定的模式序列可以用来让读者明白模式语言是怎样使用的，能怎样使用以及将会怎样使用。总而言之，一系列的模式序列可以被看作使用某一特定模式语言的指导，或者反过来，一系列的模式序列又可以作为模式语言的基础。

因此，模式序列有扮演多种角色的潜力。除了以模式故事的形式之外，模式序列通常不会直接作为展现模式语言的一部分，因而在设计模式社区中关于模式序列的讨论比模式分类及特定描述的讨论要更多。考虑到不同的模式序列反映了设计中的不同片断，并为不同场景提供不同的属性，因此记录模式序列，即使只是简要地描述，也是值得的。当然，即使是对一个规模很小且很简单的结构语言，要列举其所有合理的序列——不管是对作者还是想把它当作指南的读者来说，也都是不可能完成的任务。

实践中阐明模式语言的常见手段是模式故事。然而用故事来描述它可能有过于平铺直叙的风险，模式的例子可能被误解为模式本身，从而导致模式故事反客为主，最终侵占了模式语言本身的核心意义。尽管读者可以进行概括，但他们可能最终只注意到示例中描述的特定情况，而不会注意到它的一般性主题和结构。

因此，在一般性和特殊性之间保持适当的平衡就显得至关重要，这样才可以确保模式语言中的每个模式得到充分完全的描述，并明显强调它们之间的联系。单一模式经常可以被用在一系列不同情景和不同模式语言中。为了专注于在某一语言的角色，可以只记录与语言相关的某些特性，而减少甚至省略与其他情形相关的特性。同时模式的背景也可以被限定到只与模式语言中的前一个模式有关。这样把特定的例子、背景相关的模式以及模式之间的关系结合起来，就可以提供一种展现和使用模式语言的可行方法。

1.5 模式的连接

我们不能低估单一模式的价值，但是我们更应该重视把它们联系起来作为一个整体所带来的巨大价值。模式是外向的、喜欢结伴的、社区性的。

模式之间的网状联系反映了设计的本性。根据其扮演的角色不同，不同模式之间综合、重叠、强化和平衡的概念可能与我们最初的直觉不同：在特定设计中，基于特定代码单元的组合并不一定能够最好地反映了设计的历史、原理以及未来。



分布式系统就是这样一种系统，系统中一个你甚至都不知道的计算机出了故障却可能导致你自己的计算机不可用。

——Leslie Lamport（美国著名计算机科学家，LaTeX作者）

分布式系统是一种计算系统，系统中多个组件通过网络通信的方式互联。自从上世纪九十年代中期，因特网和万维网的爆炸性增长使得分布式系统从传统的应用领域，例如工业自动化、国防和电信，扩展到了几乎所有的领域，包括电子商务、金融服务、医疗保健、政府部门，以及娱乐业。本章描述了开发分布式系统的关键特征及其挑战，同时也介绍了为应对这些挑战而出现的几种关键软件技术。

2.1 分布式的优点

绝大部分传统软件是运行于单机系统之上的，它们的用户界面、应用的业务流程以及持久化数据都会驻留于同一台使用总线或电缆来连接外部设备的计算机上。不过，现今备受关注的系统中，几乎没有哪个还保有这种设计。如今，大多数计算机软件都运行在分布式系统中，其交互界面、应用的业务流程以及数据资源存储于松耦合的计算节点和分层的服务中，再由网络将它们连接起来。

图2-1描述了一个仓库管理控制系统的三层分布式架构，我们会在本书的第二部分详细描述该基于模式的设计，示例中的三层是通过Broker (137) 架构连接起来的。

以下特性是分布式系统之所以成为信息和控制系统[Tran 92]基础的关键。

- 协作与互联。分布式系统的重要作用之一就是能够为我们整合大量地理上分散于各地的信息和服务，如地图、电子商务网站、多媒体、大百科全书等。互联网上即时通讯系统和聊天室的流行凸显了分布式系统的另一个重要作用：与家庭、朋友、同事乃至客户保持联系。
- 经济性。包括PDA、笔记本、台式机和服务器的计算机网络通常比集中式的大型机具有更高的性价比。例如，它们支持非集中式的、模块化的应用，这使得共享像大容量的文件服务器、高清晰度的打印机等昂贵的外设成为可能。相应地，我们可以将选定的应用组件和服务分配到具有特殊处理属性的计算机节点上执行，如具有高性能磁盘控制

器、大容量内存或增强的浮点运算能力的节点，而那些简易的程序就可以运行在相对廉价的硬件上。

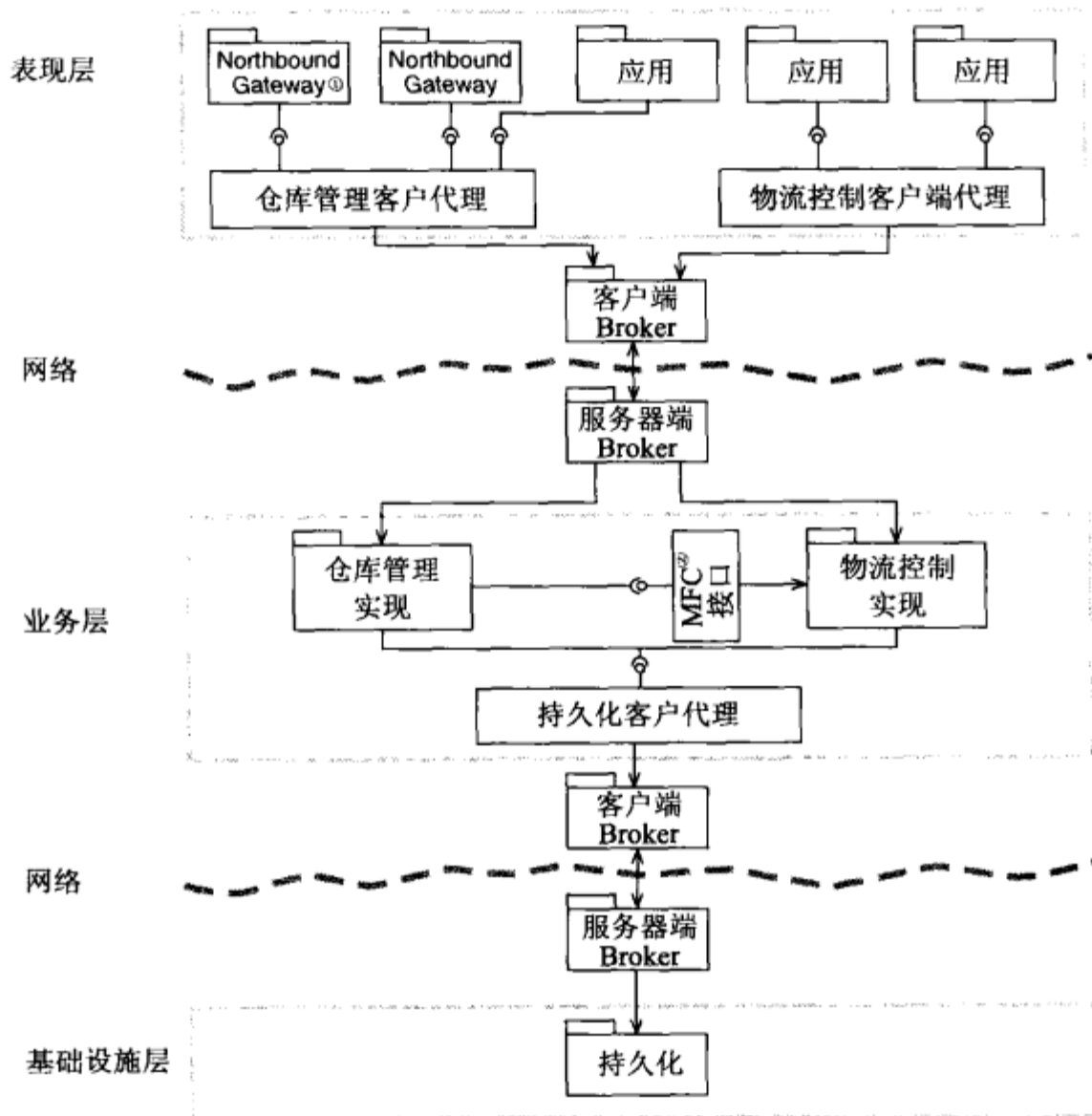


图 2-1

- **性能与可伸缩性。**成功的软件通常会随着时间推移而拥有更多的用户和需求，因此分布式系统的性能能否扩展以处理不断增加的负荷至关重要。我们可以将已联网计算节点的处理能力组合起来以获得性能的极大提升，此外，至少理论上来说，多处理器和网络更容易扩展。例如，多个计算和通信处理任务可以并行运行在数据中心的不同节点上，或运行在同一服务器的不同虚拟机上。
- **容错性。**分布式系统的关键目标之一是允许系统发生部分故障。例如，尽管网络中的所有节点都运转正常，但网络本身可能会出现故障。类似地，网络中的某个终端系统，或多处理器系统中的某个中央处理器可能崩溃。这些故障应当能被很好地处理而不影响其

① Northbound Gateway，有的地方译为北向网关，表示向高层提供的较为粗粒度的接口，本书保留原文不译。

——译者注

② 此处MFC是“Material Flow Control”的缩写，表示物流控制。——译者注

他不相关的部分或整个系统。一种常见的容错实现方式是在多个节点或网络上提供重复的服务。冗余性有助于将单节点失败的影响控制在最小范围内，它可以显著提高系统在出现部分故障时的可靠性。

- 内在的分布性。某些应用天生就是分布式的，如电信管理网络（TMN）系统、跨区域部门的企业级业务系统、对等网络（P2P）的内容共享系统、企业之间（B2B）的供应链管理系统等。分布式在这些系统中已经不再是一个可选项，而是满足客户需求至关重要的部分。

2.2 分布式的挑战

尽管分布式系统日益普及而且越来越重要，但开发人员仍需面对一系列的挑战[POSA2]，包括如下这些。

- 固有的复杂度。源于分布式系统基础原理的挑战。例如，分布式系统的组件通常驻留在不同节点的独立地址空间中，因此，与集中式系统不同，分布式系统节点间通信需要采用不同的机制、策略以及协议。此外，在分布式系统中，同步和互相协作也更加复杂，这是因为组件间可能并行运行，而网络通信则是异步的且具有不确定性。连接分布式系统中不同组件的网络会引入额外的影响因素，如延迟、抖动、瞬时失效以及过载，这些都会对系统的效率、可预测性和可靠性产生相应的影响[VKZ04]。
- 附加的复杂度。其主要来自于软件工具和开发技术的局限性，如不可移植的编程API以及不成熟的分布式调试器。具有讽刺意味的是，很多附加复杂度的引入源于开发人员的有意选择，他们更倾向于使用那些在分布式系统中难于扩展的底层语言 and 平台，如C和基于C的操作系统API和库。随着应用需求复杂度的上升，新的分布式基础架构随之出现并被发布，但它们当中的某些并不成熟可靠，这使得开发、继承和升级可用系统变得更加复杂。
- 方法和技术上的不足。流行的软件分析方法和设计技术 [Fow03b] [DWT04] [SDL05] 主要关注于构建那些“尽力而为”满足QoS需求的单进程、单线程应用。而开发高质量的分布式系统——特别是那些有严格性能要求的系统，比如说视频会议或航空控制系统——往往被留给那些有经验的软件架构师和工程人员来完成。此外，获取开发分布式系统的经验不仅需要花费大量的时间去斟酌那些与平台相关的细节，还需要通过反复尝试的方法来修正出现的错误。
- 对核心概念和技术持续的重新创建和重新发现。在软件产业的历史上，人们经常会为已解决的问题重新创建完全不兼容的解决方案。目前至少许多通用或实时操作系统可以管理同样的硬件资源。类似地，还有许多互不兼容的操作系统封装库、虚拟机和中间件，以提供存在细微差别的API接口，这些接口实现了本质上基本相同的功能和服务。如果我们将精力集中用于改进一小部分的解决方案，那么分布式系统的开发人员就可以通过重用通用工具、标准平台及组件的方式来进行快速革新。

2.3 用以支持分布式的技术

为了应对上述挑战,发展出了3个层次的支持技术:Ad hoc网络编程(ad hoc network programming)、结构化通信以及中间件[Lea02]。在Ad hoc网络编程这个层次,主要依靠进程间通信(interprocess communication, IPC)机制如共享内存、管道以及套接字(Socket) [StRa05],以便分布式组件之间互联和信息交换。IPC机制有助于解决分布式系统的一个关键问题,使得不同地址空间的组件能够互相协作。

开发分布式系统时,如果只使用Ad hoc网络编程的话也会有相应的缺点,比如说,在应用代码中直接使用套接字会导致代码与套接字API紧耦合,这时向另一种IPC机制进行代码移植,或者重新部署到其他节点上就需要较大的开销。即便是将代码移植到同一操作系统的不同版本上,也可能由于各平台IPC相关的API的细微变动而导致对代码作出相应的修改[POSA2] [SH02]。另外,直接使用IPC机制编程还会导致范式不匹配,例如,本地通信使用面向对象的类和方法调用,而远程通信却采用面向功能的套接字API和消息传递。

某些应用及其开发人员可以接受Ad hoc网络编程的上述缺陷,如汽车引擎控制器和电力网这样的传统嵌入式系统,它们运行在同构分布式环境中,其初始需求、组件配置以及IPC机制的选择很少变化。不过,绝大部分应用难以接受Ad hoc网络的上述缺陷,因为它们既需要运行于异构计算环境中,又需要面对持续变化的需求。

对分布式计算下一个层次的支持是结构化通信,它通过提供较高层次的通信机制,避免了应用代码与底层IPC机制的直接耦合,从而解决了Ad hoc网络编程的限制。结构化通信封装了与机器相关的细节,比如比特、字节以及二进制读写等。应用开发人员可以基于其所提供的编程模型来具体实现更贴近于应用领域的数据类型和通信方式。

迄今为止,最有名的结构化通信示例是远程过程调用(Remote Procedure Call, RPC)平台,比如Sun RPC[Sun88]以及分布式计算环境(DCE) [RKF92]。RPC平台允许分布式应用像在本地环境一样互相协作:一个程序可以调用另一个程序的函数,传递调用参数以及接受函数调用的结果。RPC平台隐藏了底层的操作系统API和IPC机制的细节。此外,还有一些其他结构化通信的例子,如PROFInet[WK01]为工业自动化系统提供运行时模型,它定义了几种面向消息的通信协议;ACE[SH02][SH03]则提供了可重用的C++包,它实现了不同操作系统间通信的通用框架。

尽管相对于Ad hoc网络编程而言,结构化通信已有所改进,但它并没有完全解决上一节中所描述的分布式系统面临的挑战。特别是在应用结构化通信时,分布式系统的组件仍然需要关注通信另一方的距离特性,有时甚至还要关注对方在网络上的具体位置。虽然位置信息已经能够满足某些特定的分布式系统,如静态配置的组件的部署很少改变的嵌入式系统,但是对于下面这些更复杂的分布式系统,结构化通信就不太适用了。

- 位置独立的组件。理想情况下,在分布式系统中客户端访问本地服务与访问远程服务所采用的编程模型应该是相同的。要实现这种位置独立性,我们需要将客户端和服务应用代码中处理位置(远程或者本地)细节的部分剥离出来。当然,在这种情况下分布式系统会有一些故障模式是本地系统所没有的[WWWK96]。

- 灵活的组件（重新）部署方式。随着硬件的升级、新节点的引入或新需求的增加，原有的应用服务在网络节点中的部署方式可能不再是最优的，因此需要重新部署分布式系统的服务。理想情况下，这时应当不会破坏原来的代码，而且不需要关闭整个系统。
- 集成遗留代码。很多时候，我们不会从头开发复杂的分布式系统，相反，大多数时间我们是在已有组件或应用的基础上进行开发，而这些原有组件最初可能并不是针对分布式环境而开发的——实际上我们可能连源代码都没有。继承遗留代码主要是为了利用已有的软件组件，降低软件认证开销或减少投入市场所需要的时间。
- 异构组件。分布式系统正越来越多地面对将不同的企业级分布式系统组合起来的任务，这些系统通常基于不同的现有技术，而不仅仅是将内部开发的专有软件集成起来。此外，随着企业应用集成（Enterprise Application Integration, EAI）[HoWo03]的出现，我们必须将不同语言开发的组件和应用集成到一致的分布式系统中。一旦集成起来，这些特性各异的组件应当能够互相配合并正确地执行日常任务。

上述这些挑战已经不是结构化通信所能解决的了，它需要专门的中间件[ScSc01]驻留在应用和底层的操作系统、网络及数据库之间，提供分布式系统所需的基础设施功能。中间件提供了上述所有特性，从而使得应用开发人员可以集中精力，关注他们的首要任务：实现应用领域相关的功能。

在认识到中间件拥有广阔的市场需求后，微软、IBM、Sun以及OMG、W3C等公司和组织纷纷开始发展分布式计算相关技术。下面我们将介绍一些流行的中间件技术，包括分布式对象计算中间件、组件中间件、发行/订阅中间件、面向服务架构（SOA）以及Web服务 [Vin04a]。

2.3.1 分布式对象计算中间件

分布式对象计算（Distributed Object Computing, DOC）中间件技术出现于20世纪80年代后期和90年代初期，它对当时的分布式系统开发有重要贡献。分布式对象计算中间件技术代表了两种主要信息技术的结合：基于RPC的分布式计算系统和面向对象的设计与编程技术。开发基于RPC的分布式技术，比如DCE[OG94]，主要关注于如何将多台计算机集成起来，以作为统一的可扩展计算资源来使用。类似地，面向对象的系统开发技术则利用经典的设计模式和软件架构，通过创建可重用的框架和组件来降低复杂度。因此，分布式对象计算中间件可以利用面向对象技术灵活有效地分发多个、通常是异构的计算和网络元件，以提供可重用的服务和应用。

用来构建分布式系统应用的CORBA 2.x [OMG03a] [OMG04a]和Java RMI [Sun04c]是分布式对象计算中间件技术的典范。它们通过在客户端和服务端之间定义接口规范，以便客户端可以调用服务器所提供的对象服务而不必知道其具体位置。标准的分布式对象计算中间件技术，比如CORBA，还允许自定义通信协议以及对象信息模型，这使得由不同语言编写的在不同平台上运行的各种应用能够相互协作[HV99]。

虽然分布式对象计算中间件的技术成熟、性能稳定并且功能强大，但它还有如下一些不足。

- 缺少功能边界。CORBA 2.x和Java RMI对象模型把所有接口都理解为客户端与服务端端的约定，不过，它们并没有提供一种标准的装配机制来避免相互协作的对象在实现时存在

相互依赖。例如，若某个对象的实现依赖于另一个对象，那么它需要明确地发现并连接到被依赖对象上，为此，开发人员必须显式设计某些用于关联服务和对象接口之间的连接，这不仅增加了额外的工作，同时也使得最终实现既脆弱又难以重用。

- 缺少软件部署和配置标准。在分布式对象计算中间件中没有一个标准化的方式去实现远程发布和启动对象。因此，在实现此功能时，应用管理员必须首先使用内部脚本和过程来将软件实现分发到目标机器上，然后配置目标机器和待执行的软件实现，最后再实例化软件。唯有如此，它们才能为客户端所用。除了上面一种情况外，软件实现还会经常被修改以适应Ad hoc部署机制。大多数可重用软件与其他软件的交互需求会进一步加剧此问题的严重性。缺少高层次的软件管理标准直接导致了系统的可维护性和软件的可重用性大幅降低。

2.3.2 组件中间件

为解决上述分布式对象计算中间件技术的缺陷，20世纪90年代中后期出现了组件中间件技术。值得一提的是，为了解决缺少功能边界的问题，组件中间件技术允许一组高内聚的组件对象通过提供或请求接口的方式来实现交互，同时它还定义了普通应用服务器上执行这些组件对象所需要的标准运行时机制。为了解决缺少标准组件部署和配置机制的问题，组件中间件通常还详细描述了分包、定制、组装和在分布式系统中分发组件的基础机制。

EJB[Sun03][Sun04a]和CORBA 组件模型 (CCM) [OMG02][OMG04b]是组件中间件技术的典型代表，它们定义了以下一些基本角色及其关系。

- 组件是功能实现的实体，它暴露了一套命名接口和连接点，以便组件之间可以互相协作。命名接口是指那些其他组件可以同步调用的服务方法。连接点与其他组件所提供的命名接口相联合，就可以建立客户端到服务器端的关联。有些组件模型还提供事件源和事件接收器，以实现异步消息传递。
- 容器为组件实现提供了服务器运行时环境。容器包含了各种预定义的钩子函数和操作，这些内容使得组件可以访问到相关的策略和服务，如持久化机制、事件通知、事务、复制 (replication)、负载均衡以及安全机制。每个容器都定义了运行时策略的集合，包括事务、持久化、安全以及事件分发策略等。同时，容器还负责初始化托管组件并为其提供运行时环境。组件的实现通常使用XML格式的关联元数据来指定所需的容器策略 [OMG03b]。

除了上述要素之外，组件中间件技术还利用说明性的元数据为开发过程的各个阶段提供全面的信息[DBOSG05]，进而为软件开发生命周期各个阶段的不同方面带来了自动化：组件实现、分包、装配以及部署。这些特点使得组件中间件技术能够以比分布式对象计算中间件技术更快更健壮的方式创建应用。

在组件架构中，定义良好的关联关系亦存在于组件与对象之间[Szy02]。一般而言，组件在构造时创建，可能在运行时载入，并定义了运行时行为的实现细节。同样地，对象在运行时创建，其类型在组件内封装，它们运行时的动作将最终决定程序的行为。由于存在这样的关联关系，组

件能够被定义、构造和加载，而对象也会被创建并发生交互。

2.3.3 发布/订阅中间件和面向消息的中间件

RPC平台、分布式对象计算中间件以及组件中间件技术，都是基于请求/响应的通信模型，客户端向服务器发出请求，服务器将响应发送回客户端。然而，对于某些类型的分布式应用，特别是那些需要对外界刺激和事件做出反应的系统，如控制系统和在线股票交易系统，请求/响应通信模型在某些方面就不大适用了。这些不适用的方面包括：采用客户端和服务器的同步通信方式时，模型不能充分利用网络和终端系统的并行输入能力；采用指定的通信方式，客户端必须知道服务器的标识，这会导致其与特定的接收方紧耦合；如果使用点对点的通信方式，则客户端同一时刻只能和一个服务器通信，从而限制了客户端将信息传递到所有需要信息的接收方的能力。

正因为如此，在某些分布式系统中采用了备选的结构化通信方案，即采用面向消息的中间件技术或者发布/订阅中间件，前者包括IBM的MQ系列[IBM99]、BEA的Message Q[BEA06]、TIBCO的Rendezvous等，而后者则包括Java消息服务(JMS)[Sun04b]、数据发布服务(DDS)[OMG05b]、WS-NOTIFICATION[OASIS06c]等。面向消息的中间件技术主要优点在于支持异步通信：发送方在传送数据给接受方后不需被阻塞以等待接收方的响应。很多面向消息的中间件平台提供了事务机制，消息能被可靠地排队或存储，直到接收方把它们取出来。发布/订阅中间件增强了这种异步通信的能力，它不需要通过传送事件数据来明确指定接收方地址，此时发布方和订阅方是松耦合关系，双方甚至都不需要知道彼此的存在；发布/订阅中间件还具备了组播的功能，可以实现多个订阅方同时接收同一个发布者发布的信息。

发布/订阅中间件技术的特点是允许运行在独立节点的应用，从分布式系统的一个全局的数据空间读/写事件消息。应用需要公开它们希望产生的事件，就可以利用这个全局数据空间与其他程序分享信息；而另一些应用则可以通过声明自己感兴趣的主体，或者简单地处理队列中的所有事件，就能够收到所有相关的事件消息。

发布/订阅中间件技术主要包含以下角色。

- 发布方是事件源，它们产生要传播到系统的关于特定主题的事件消息。根据实现架构的不同，发布方可能需要描述所生成优先级的事件的类型。
- 订阅方是系统的事件接收方，它们使用自己感兴趣主题的数据。某些实现架构要求订阅方声明它们所希望接收信息的过滤信息。
- 事件通道是系统中将事件从发布方传播到订阅方的组件。它们可以把事件传播到分布式系统的远程订阅者那里，还可以完成各种服务，比如过滤及寻址功能、质量保证增强功能以及错误管理功能。

从发布方传递到接收方的事件可以有多种表现形式，从简单的纯文本消息到类型丰富的数据结构体等。相应地，用来发布和订阅事件的接口可以很通用，比如用来交换WS-NOTIFICATION中任意动态类型的XML消息的send和recv方法；也可以是特定的，比如DDS中用于交换静态类型事件数据的数据读/写方法。

2.3.4 面向服务架构和 Web 服务

面向服务架构 (SOA) 是一种组合和使用那些由不同组织或个人所控制的分布式处理能力的方式。它提供了一种统一的方式来提供、发现、交流和使用分布式系统能力,让这些松耦合的软件服务互相协作以满足业务处理和用户的需要[OASIS06a]。“SOA”一词起源于20世纪90年代中期[SN96],它是对当时所用的中间件互操作性标准的一种总结,包括RPC、ORB以及基于消息的平台。

万维网的普及以及对早期中间件技术中经验的总结产生了最初版本的SOAP协议[W3C03]。SOAP是在计算机网络中可用来交换基于XML[W3C06b]消息的一种协议,它通常使用HTTP[FGMFB97]作为传输机制。SOAP协议的设计初衷是使其成为一种平台无关的协议,从而支持网络上各种不同类型中间件的协作,包括CORBA、EJB、JMS以及各种专有的面向消息的中间件系统,如IBM的MQ系列和TIBCO Rendezvous。

SOAP的引入导致了SOA的一种变体Web服务 (Web Service)——正在由World Wide Web Consortium (W3C) 进行标准化——的流行。Web服务允许开发人员将应用逻辑封装到服务中,其接口通过Web服务描述语言 (WSDL) [W3C06a]来描述。基于WSDL的服务通常通过标准的因特网高层协议 (比如HTTP基础上的SOAP) 来访问,Web服务可以用来构造企业服务总线 (ESB),这是一种简化独立系统之间交互的分布式计算架构。Mule[Mule06]和Celtix[Celtix06]都是ESB方案的开源案例,它们可将不同类型的系统融合到统一的分布式应用中。

尽管存在一些公认的缺陷[Bell06][Vin04b],Web服务还是成为了绝大多数企业级应用的首选技术。然而这并不能说明Web服务将会取代早期的中间件技术,如EJB和CORBA。相反,Web服务是这些早期成功的中间件技术的补充,并提供了互操作性的标准机制。典型的产品包括微软公司的Windows通信基础设施 (Windows Communication Foundation, WCF) 平台[MMW06],由IBM、BEA、IONA定义的服务组件架构 (Service Component Architecture, SCA) [SCA05],以及其他组件开发和Web技术的组合。如同组件一样,WCF和SCA平台提供了黑盒功能,这样就可以描述和重用功能而不必关心这些服务是怎样实现的。不过,不同于与传统的组件技术,对WCF和SCA平台的访问不能通过对象模型相关的协议 (如DCOM[Box97][Thai99]、Java RMI和CORBA),而对Web服务的访问要通过像HTTP和XML这样的Web协议和数据格式来完成。

最初的Web服务提供了一个基于HTTP交换XML消息的RPC模型,那时起Web服务就宣称将取代更复杂的EJB组件或CORBA对象的机制。不过,由于Web服务基于纯文本的协议,如基于HTTP的XML[EPL02],对于那些细粒度的分布式资源访问,其性能通常比分布式对象计算中间件技术要低几个数量级,因此在性能优先的应用场合,如航空、军事、财务以及流程控制领域的分布式实时嵌入式系统,对Web服务的应用远远不如在那些松耦合的面向文档的场合 (如供应链管理系统) 中用得更多。

今天的Web服务技术并未试图取代原有技术,而是更多地关注于与中间件的集成,为现有的中间件平台提供更多的价值。WSDL允许开发人员抽象地描述Web服务的接口,同时定义具体的协议和传输绑定,以便在运行时能访问到这些服务。通过在不同的中间件平台间提供通用通信机

制，Web服务允许组件在整个组织的全部应用集合中重用，而不受其具体实现技术的影响。例如，像Apache Web服务调用框架（Web Service Invocation Framework, WSIF）[Apache06]、Mule、CeltiXfire等都致力于通过EJB、JMS或SCA来透明访问Web服务。这种向集成方面的转移使得这些由不同技术实现的服务能够被集成到ESB中，从而可以为广泛的客户应用所访问。因此在可以预见的将来，中间件集成将成为Web服务的核心关注点[Vin03]。通过专注于集成，Web服务在增加重用性的同时还降低了中间件的束缚，它还允许开发人员使用适当的中间件技术以满足自身的需求，而不必关心如何与现有系统协作。

2.4 中间件技术的局限性

尽管本章介绍了中间件技术的很多益处，但是对于分布式系统而言，它并不是万能的。上面所描述的中间件技术主要充当了分布式系统中不同组件之间的“信使”，而且即使做出很大努力，有时它也不能保证成功传送。因此，分布式系统必须随时准备处理网络故障以及服务器宕机等情况。此外，中间件也不能神奇地解决由错误的部署决定所引入的问题，而这些问题会严重降低系统的稳定性、可预测性和扩展性。

换言之，中间件技术是分布式系统的重要组成部分，但它并不能处理那些超出其范畴且与特定应用有关的任务。尽管中间件技术允许分布式系统与其组件的具体位置无关，但在设计和验证系统时还是需要仔细斟酌。

语言的极限就是世界的极限。

——Ludwig Wittgenstein（奥地利哲学家、数理逻辑学家）

本章开始介绍分布式计算的模式语言，勾画了其意图、范畴、对象、起源、产生、结构、内容、表现以及应用。由此，本章展现了模式语言的背景信息，描述了它与常规软件工程实践以及现有模式文献的关系。

3.1 意图、范畴和对象

分布式计算模式语言的主要意图是对分布式软件系统关键领域的最佳实践和先进经验进行概括、介绍，并充当这些实践和经验的学习指南和沟通工具。这门语言所包含的既有应用分解、组件部署、通信中间件这些根本性和战略性的内容，又有分布式系统中组件的设计细节和系统资源管理这类辅助性、战术性的内容。

为了达到这一目的，模式语言把不同来源的各种模式连接成一个单一内聚的模式网络，提供创建分布式软件的全面而一致的视角。模式语言记录了目前构建分布式软件系统的产品经验——因此我们很有信心将它展现在这里，但这并不是分布式架构的最终结论。我们希望描述的是基于实践经验的模式，而并非研究它将来的演化。

软件架构师、开发人员以及高校学生，可以使用我们的分布式计算模式语言去创建、交流和重构分布式系统，同时也可以通过模式语言更好地理解通用中间件平台和产品的基线架构和范式。此外，产品经理和项目经理可以深入理解他们所领导开发的分布式系统的核心功能，并且更加便捷地和软件架构师及开发人员进行交流。然而，我们并不打算让最终用户或顾客直接使用我们的模式语言。虽然我们可以运用现实世界的比喻来让他们接受模式语言，但这需要换一种表达方式。

总体而言，我们的模式语言并非包罗万象的教程，它主要关注于分布式软件系统的设计，因此我们假定读者对于核心的分布式计算概念和机制有一定程度的了解，如死锁、事务、同步、远程以及任务切换。

3.2 起源

我们模式语言所包含的模式源自很多软件专家，包括Deepak Alur、Bruce Anderson、Kent

Beck、Roy Campbell、Jens Coldewey、John Crupi、Eduardo Fernandez-Buglioni、Martin Fowler、Erich Gamma、Richard Helm、Michi Henning、Gregor Hohpe、Duane Hybertson、Prashant Jain、Ralph Johnson、Wolfgang Keller、Michael Kircher、Doug Lea、Silvano Maffeis、Dan Malks、Gerard Meszaros、Regine Meunier、Hans Rohnert、Alexander Schmid、Markus Schumacher、Peter Sommerlad、Michael Stal、Steve Vinoski、John Vlissides、Markus Völter、Eberhard Wolff、Bobby Woolf、Uwe Zdun，当然还有我们自己，因此你可以认为这门语言是专家们对构建分布式软件系统的总结性表达。

尽管模式语言中的大多数模式——以及它们之间的一些关系——已经存在，但很难将它们联系成一个内聚、一致且更广泛的模式语言。正因为如此，我们不得不重新描述所有的模式，一方面是为了更明确地强调其本质，另一方面也是为了使它们更适合本书的背景。例如，我们必须找出这些模式在分布式系统中能解决的问题、怎样解决这些问题、为什么这样去解决以及模式之间的关系如何。对大部分模式而言，这些信息存在于原有的模式描述中，我们只需要将它们提取出来。不过，对于其他一些模式，我们必须更深入地挖掘我们自己和其他人的经验。只有很小一部分模式是“新的”：它们主要用于覆盖那些已有模式没有描述的东西，或者作为某个公共主题之下一系列已有模式集合的“保护伞”。

我们对很多模式作了改进，以适应分布式计算的特殊环境，我们还有针对性地补充了关于驱动因素及其效果的讨论并在模式之间引入了新的关系，因此与原有模式相比，它们能更好地与分布式计算环境联系起来。然而，为了使读者更好地关注这些模式所构成的完整画面，我们有意省略了原有模式描述的部分细节，改用较少的篇幅来描述。例如，我们省略了核心角色的CRC卡等。如果你对更细粒度的结构、交互图、实现要点以及活动、变化、示例和已知问题及结果感兴趣的话，可以参考模式来源，那里我们列出了模式语言中所用到的所有模式。

3.3 结构和内容

我们关于分布式计算的模式语言一共包含114个模式，分为13组问题域。每个问题域描述一个和构造分布式系统相关的特定技术主题，并包含模式语言中针对该问题域的所有模式。问题域的目的是为了使得模式语言及其模式更容易接受和理解：将解决相关问题的模式放在一个公共且范围明确的环境中进行讲解与讨论。问题域依照其在构造分布式系统时的相关性和可用性的顺序可以分为以下13类。

(1) 从混沌到结构

该问题域包括分布式模式语言的基础模式。它们帮助我们从需求和限制的混沌中得到粗粒度的软件结构，进而产生清晰而独立的实体部件以构成整个要开发的系统。此外，本章中的模式还从操作方面（如性能和可用性）到开发质量（如可扩展性和可维护性）讲述了可持续软件架构的几个关键概念。

(2) 分布式基础设施

该问题域描述了中间件相关的模式。中间件是分布式软件的基础设施，帮助我们简化分布式系统中的应用开发。这里的模式可以帮助开发人员理解常用中间件产品和平台所支持的基础通信

范式，以及其软件架构的关键方面。

(3) 事件分离和分发

不管应用使用一个多么复杂的通信模型，如同步请求-响应、异步消息或发布/订阅传播机制，分布式计算的核心仍然是处理和响应网络中接收到的事件。事件驱动的内核扮演着至关重要的角色，正因为如此，它绝不能成为系统的性能瓶颈。

(4) 接口划分

接口是一个组件的“名片”，客户端可以通过它来了解组件的功能和使用协议。它们应当方便客户端和组件之间进行正确而有效的协作，因此，设定可用性强、有价值的组件接口对软件开发的成功至关重要。不过，要做到这一点是很困难的，因为接口必须能够清楚地反映组件的责任且对客户端有意义，另一方面，组件实现发生变化和改进时不能影响到客户端。

(5) 组件划分

组件是基本实现单元，它为客户端提供定义清晰的服务。虽然客户端通常并不关心组件的内部设计，但是组件的划分会严重影响其对外部可见的质量特性，如性能、可伸缩性、灵活性、可用性以及容错性。

(6) 应用控制

首先将应用程序的用户输入转换成具体的功能服务请求，然后执行这些请求，最后再将结果转换成对用户有用的输出，这一过程本身是很难实现的。不过，将应用程序的用户界面和其功能实现隔离开似乎更为困难。这种隔离通常是为了方便用户界面和应用功能的各自改进以便更好地适应底层技术的变化，或者是为了能够在不同平台上部署不同的组件配置。

(7) 并发

分布式系统软件经常能从并发中受益，尤其是那些需要同时处理多个客户请求的服务器和服务端的应用。此外，越来越多的多核CPU和多CPU计算机被设计出来，它们并行执行多个控制线程以弥补相对于摩尔定律的差距[Sut05a]，因此分布式系统的开发人员必须精通进程和线程管理机制。没有一种软件并发架构可以适用于所有平台和各种负荷情况。

(8) 同步

同步访问共享组件、对象和资源，并避免死锁、竞争和其他并发问题是构建分布式系统最困难的任务之一。此外，同步会产生很大的开销，因此设计应用程序时应尽量减少或避免不必要的同步。

(9) 对象交互

在独立程序中，对象间的协作主要包括互相调用方法和服务、传递调用参数以及同步等待被调用对象返回结果。然而，为了平衡各种相互冲突的驱动因素，如延时、可伸缩性和可靠性等，分布式系统中的交互经常要复杂得多。

(10) 适配与扩展

有些应用程序只为某个特定客户开发，而另一些则是针对大众市场开发的产品。即使是面向单一客户的应用也能从公共架构的基础上获益，这样可以简化重复业务，在新客户提出相似功能需求的时候能简化定制工作。多个用户应用同样也能从某一特定软件基础设施上受益，但是，不

同用户经常会有不同的需求，而且系统默认情况下并未提供支持。因此，会长期存在的分布式系统中的组件应当是可配置的、自适应的和便于改进的。

(11) 模态行为

系统中的某些对象本质上是状态驱动的，其所有方法——或绝大部分——根据自身状态的不同会有不同的行为。我们有很多种方法来实现对象状态驱动的生命周期，有时用简单的标志和对象方法中的条件语句来控制流程就足够了。然而，有些时候，对象中大部分甚至所有的方法在不同状态时候的行为截然不同，这时候通常需要使用状态机来表示其生命周期。在实践中，开发人员经常会面对使用状态机的情形，某些情况下，这样做会导致实现上不必要的复杂性。

(12) 资源管理

资源管理对分布式系统的成功至关重要。例如，在内存中保持太多的无用对象会导致服务器的性能下降。然而，我们很难正确有效地管理资源。许多应用的服务质量特性，如性能、可伸缩性、灵活性、稳定性、可靠性、可移植性和安全性，依赖于怎样有效地创建和获取、访问和使用、销毁和释放资源，以及整体上如何管理资源。而试图平衡各种折中方案则会使得资源管理更加困难，因为满足某种需求经常会和另一种需求相冲突。

(13) 数据库访问

许多分布式系统使用数据库来存储持久化数据，越来越多的系统在使用面向对象技术的同时使用关系数据库模型。但是对象模型和关系模型之间并不能完美地互相映射。如何将面向对象的应用灵活有效地映射到关系数据库架构（relational database schema）往往比我们想象的要困难得多，并且经常会遇到超过以往的挑战。

就构造分布式系统的各技术方面而言，上述13个问题域互相补充和完善。图3-1列出了各问题域之间的连接关系。

还有一些问题域之间的关系没有在图3-1中标识出来，如有些阐述适应性和扩展性问题的模式会引用接口划分相关的模式，具体实例就是定义明确稳定的接口以隐藏实现细节。比较而言，我们明确列出的问题域间的关系要比那些省略的关系更重要。

除了问题域之外，分布式模式语言的模式之间还通过其他方式联系起来。因此我们采用自上而下的方式来描述它们，首先描述语言根模式，再描述实现根模式的那些模式，再描述实现这些模式的模式……最终描述这个网络中的“叶子”模式——实现其他模式，而本身不再被模式语言中更细粒度的模式来描述的模式。

因此我们的模式语言定义了一种类似溜溜球的过程来描述分布式系统：从基础架构的定义开始，然后描述它们的组件，最后以阐述各种组件内部设计的主题来结束，同时也支持从某一特定设计开始重构系统。因为书的内容必须是顺序的，而这种语言描述并不具有层级结构。因此，有些模式在书中描述的顺序与它在模式语言网状层级中应有的位置可能不大一致。

本书中将要模式语言中的114个模式进行明确描述，具体如下。

- 从混沌到结构：Domain Model (106)、Layers (108)、Model-View-Controller (109)、Presentation-Abstraction-Control (111)、Microkernel (113)、Reflection (114)、Pipes and Filters (116)、Shared Repository (117)、Blackboard (119) 和 Domain Object (121)。

- Command Processor (199)、Template View (200)、Transform View (201)、Firewall Proxy (202) 和 Authorization (204)。
- 并发: Half-Sync/Half-Async (209)、Leader/Followers (211)、Active Object (212)、Monitor Object (214)。
 - 同步: Guarded Suspension (221)、Future (223)、Thread-Safe Interface (224)、Double-Checked Locking (225)、Strategized Locking (226)、Scoped Locking (227)、Thread-Specific Storage (228)、Copied Value (230) 和 Immutable Value (231)。
 - 对象交互: Observer (237)、Double Dispatch (238)、Mediator (239)、Memento (242)、Context Object (243)、Data Transfer Object (244)、Command (240) 和 Message (245)。
 - 适配与扩展: Bridge (255)、Object Adapter (256)、Interceptor (260)、Chain of Responsibility (257)、Interpreter (258)、Visitor (261)、Decorator (262)、Template Method (265)、Strategy (266)、Null Object (267)、Wrapper Facade (269)、Execute-Around Object (264) 和 Declarative Component Configuration (270)。
 - 对象行为: Objects For States (274)、Methods For States (275) 和 Collections for States (276)。
 - 资源管理: Object Manager (291)、Container (288)、Component Configurator (289)、Lookup (292)、Virtual Proxy (294)、Lifecycle Callback (295)、Task Coordinator (296)、Resource Pool (298)、Resource Cache (299)、Lazy Acquisition (300)、Eager Acquisition (301)、Partial Acquisition (303)、Activator (304)、Evictor (305)、Leasing (306)、Automated Garbage Collection (307)、Counting Handle (309)、Abstract Factory (311)、Builder (312)、Factory Method (313) 和 Disposal Method (314)。
 - 数据库访问: Database Access Layer (318)、Data Mapper (320)、Row Data Gateway (321)、Table Data Gateway (323) 和 Active Record (324)。

对于某些问题域, 所有有文献记载的模式比我们明确描述的还要多。例如, 远程[VKZ04]和消息[HoWo03]机制方面的全部模式, 这是对分布式基础设施问题域模式的补充。因此分布式模式语言还集成了一些改进和补充上述模式的其他模式来源, 主要包括以下方面。

- 设计服务器组件的模式语言[VSW02]
- 远程访问模式语言[VKZ04]
- 消息传递模式语言[HoWo03]
- 设计企业级应用架构的模式集合[Fow03a]
- 安全性方面的模式集合[SFHBS06]
- 访问关系型和对象型数据库的四种模式集合和语言[BW95][KC97][Kel99][Fow03a]
- C++引用计数模式语言[Hen01b]
- 设计使用特定中间件平台的两个模式集合[ACM01][MS03]

换句话说, 我们的分布式计算模式语言并不是孤立的, 而是与现有的模式领域紧密集成在一起。这种连接关系把近150个模式引入到我们的模式语言中, 从而使其成为目前为止软件领域有记录的最大模式语言之一。尽管我们并没有在本书中详细描述这些模式, 但它们是我们的分布式

计算模式语言内在的一部分。本书因为空间有限无法将其一一列出，请参阅上面所列模式的参考文献。

此外，需要强调的是，我们并不认为我们的模式语言是完善的——它还处在不断发展中。我们并没有全面覆盖的所有问题，如安全性，而且随着构建分布式系统的经验不断累积，新的模式可能会被集成到模式语言中，原有的模式也可能需要调整。

3.4 模式的表现

为了便于阅读、理解和掌握我们分布式计算的模式语言，同时也为了便于读者汲取有用的建议来构建自己的系统，所有问题域及模式的描述都采用下面的通用结构形式。每个问题域的描述会给出问题的概述和某一特定技术主题的解决方案，同时总结了采用相关模式解决特定问题时所要考虑的问题。问题域的描述分为以下4个部分。

- 对问题域的范畴和主要挑战的简要介绍，它描述了其所属模式的一般环境。
- 摘要说明解决该问题域中挑战的模式以及这些模式是怎样集成到模式语言中的。
- 对该问题域相关模式进行比较和对比的讨论，同时指出了不属于本模式语言范畴的一些应用场景。
- 模式描述，主要集中在应用环境及其问题、驱动因素、解决方案、相应的效果、实现中的关键注意事项以及与模式语言中其他模式的关系。

图3-2描述了一个问题域的简要结构。

我们设计这样的模式格式来快捷地表现每个模式的核心内容，从而有助于读者了解这个模式是关于什么的，它如何描述问题及驱动因素，应用时应当考虑什么样的后果以及怎样使用模式语言中的其他模式来实现它。我们的目标是提供足够的细节以理解每个模式，同时又不影响它和整个模式语言的协调。我们发现能满足这个目标的最合适的模式格式与Christopher Alexander [AIS77] 所使用的格式非常接近。

我们所使用的格式的开头是模式名，模式名可以标上一颗星、两颗星或者没有星。星的数目表明了我们对这个模式成熟性的确认度。两颗星表示我们非常确认该模式针对其所在问题域描述的是一个真正的问题，而且该模式建议的解决方案或者相应的变体方案对于有效解决问题至关重要。一颗星表示该模式描述的是一个真正的问题，其方案是一个不错的解决方案，但是它还不够成熟。没有星表示我们时常会发现该模式所描述的问题，而且其解决方案也是有用的，但是该模式需要很大的改进才能达到一颗星或两颗星的标准。没有星可能还表示在这种场合下有更好的备选模式。

模式名称后面是模式应用的环境。它描述了我们能使用此模式的一种或多种开发活动，同时包含了在这些活动的实现中所要用到的各种相关模式的名称及其引用页。因此，这些引用使得该模式与那些可以从中受益的模式语言中的“高层”模式连接起来。

应用环境后面就是模式的主体部分。我们把模式的主体部分和简介部分用3个钻石符号“◆◆◆”分隔开。主体部分的第一段包含问题的本质描述，所以用黑体来描述这一段。接下来是问题的驱动因素：对解决方案的要求和期望特征是什么？解决该问题需要考虑的限制条件又是什么？每一个驱动因素通常都对应了模式问题域简介中的某个（或部分）挑战。

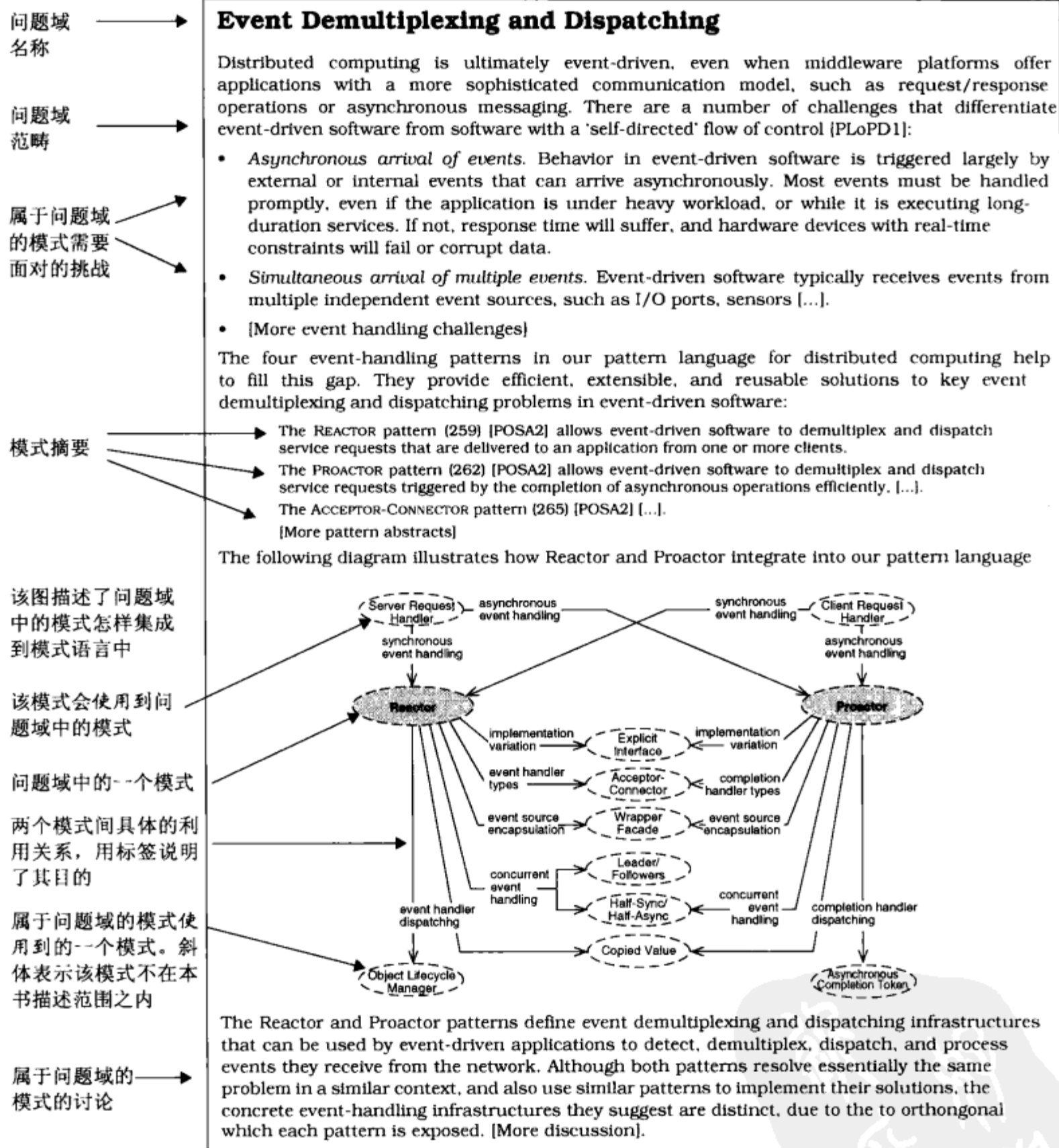


图 3-2

单词“因此”引入了模式的下一节：模式所建议的解决方案的核心部分。方案的核心部分用一段或几段黑体字作为简介，这部分可以认为是实现该模式的“迷你流程”。介绍部分的第一句或前两句强调了解决方案的基本原则，然后逐步描述具体的结构以及执行此结构的行为轮廓。我们会给出一个描述其结构和行为的图。这里，我们有意使用了与软件系统流行的建模格式不同的

标记方式。

我们这么做的原因之一是为了避免“错误的具体化”——经常导致读者误认为图上所描述的方式是实现模式的唯一正确方式。相反，我们提供了一个方案的草稿，而不是类和对象的具体规范及其关系。因此我们的标记语言混合了很多元素：角色规范、角色组织、角色协作、伪接口以及伪代码等所有展现特定模式所需要的东西。

另一个“◆◆◆”符用来表示主体部分的结束，接下来的段落更深入地解释了解决方案。例如，我们会描述模式的建议结构和行为，证明为什么这样能解决我们的问题及其驱动因素，同时我们还会列出其重要的结果。这部分的描述将该模式和模式语言中的其他模式联系起来。如果另一个模式有助于当前模式的实现，我们会引用其模式名及对应的页码，同时简要总结其在实现中的贡献。

当然，使用其他模式只是一个建议，因为应用开发中的具体环境决定了应用这些模式是否适合。使用另一个模式对解决方案可能重要，也可能不重要。在模式描述中，我们会明确区分它们的不同：当应用某个模式是必须的，我们会明确指出其描述语言也是必须的，而对其他情形我们只是提供一个建议，而不强制使用。

图3-3列出了我们使用的模式格式及其标注。

每个模式的描述可以通过下面3种方式阅读：

- 如果你只关心简要介绍，那么仅阅读记录了模式精髓的黑体段落部分即可。
- 如果你还想知道和问题描述相关的驱动因素，以及解决方案的结构和行为细节，那么请阅读“◆◆◆”前的所有部分。
- 最后，如果你对模式怎样集成到模式语言中感兴趣，请阅读整个描述部分。

因为我们更关注模式语言，而并非单一模式，所以每个模式实现的细节描述并不是我们模式语言关注的焦点。如果你对模式本身的细节描述感兴趣，可以参考相关模式的原始文献。

3.5 实际应用

本书介绍模式语言的目的是为了覆盖分布式系统的最佳实践，同时提供一些有助于构建新系统或重构现有系统的实践指导。因此在实际的软件项目中使用该语言是十分便捷的。在开发新的分布式应用时，你可以由根模式Domain Model（领域模型）入手进入该语言。该模式通过分离开发中的不同问题域及基础设施的技术考虑提供了对整个应用的基本划分方式。然后你可以按照领域模型中的实现指导在开发系统中一步一步地引入模式语言中的其他模式，比如Layers (108)、Domain Object (121)、Broker (137) 和Model-View-Controller (109) 等。这些模式有助于解决一些子问题，比如分布式基础架构、应用和组件划分以及在实现领域模型过程中出现的内部组件设计问题。

被引用的模式通常按照它们出现的顺序应用，除了那些替代性模式之外。相应地，当实现被引用的模式时，你可能会参考对当前设计提供更细节性帮助的其他模式。这是一个递归的过程，直到不需要引用其他模式的模式实现为止，或者你决定不按照参考的模式实现为止。因此，本书关于模式语言的描述顺序在很大程度上取决于我们认定的主要模式顺序。

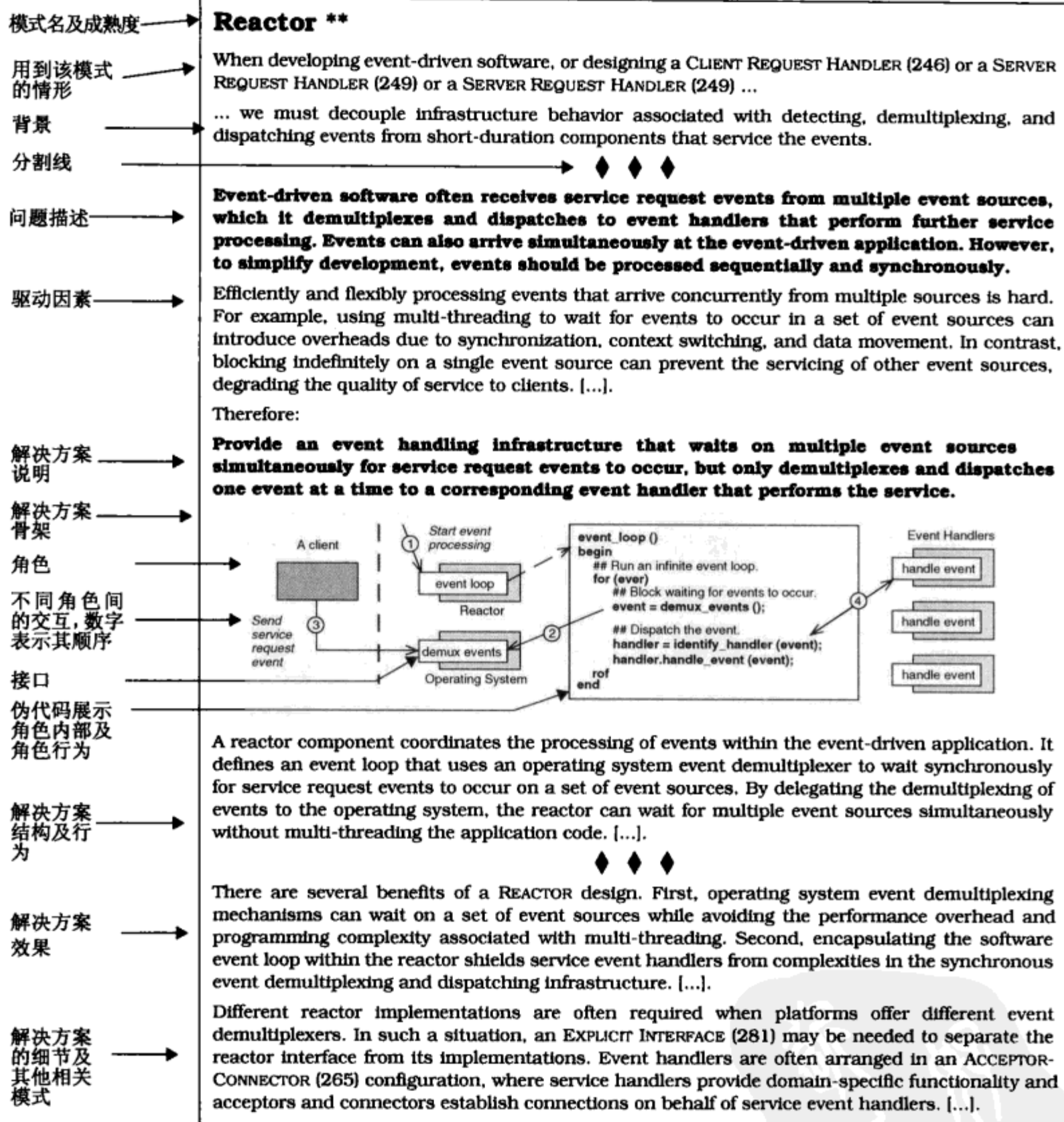


图 3-3

你可以采用“广度优先”或者“深度优先”的方式来遍历始于领域模型的所有模式，或者结合两种方式。其结果将是一系列指导设计开发分布式系统的模式。因此应用这个模式序列的软件架构会表现为一系列紧密集成的模式，这些模式互相完善、互相补充，一致而内聚地结合在一起。与此类似，当重构一个已有分布式系统的设计时，你可以从要重构的问题所对应的特定模式开始，

应用接下来的相关模式。

使用模式语言允许你创建近乎无限类型的独特分布式系统软件结构。不同的需求、设计目标以及先决条件都可能会导致你选择某个特定模式来取代我们模式语言中所建议的模式，或者遵循，或者放弃某一特定的参考模式实现。每一个不同的决定都会产生模式语言中的一条不同路径——因此会有一个不同的模式序列——从而产生一个不同的软件架构。我们的模式语言承认并支持以下事实：不存在适用于所有场合的分布式软件架构。尽管如此，根据我们的模式语言创建的所有软件架构和成功的分布式系统都有着一致的原理和风格。

为了说明分布式计算模式语言是怎样应用于产品开发中，我们会在第二部分模式故事中深入描述模式语言中的模式序列是如何形成一个仓库管理控制系统的架构。该部分介绍了仓库管理的范畴并概述了一个对应的“领域模型”，在后续章节中，它将其逐步转化为一个具体的软件架构。我们从这个系统基线架构开始，检查其通信中间件的内部实现，最后描述表示仓库储存拓扑结构的子系统。每一节都描述了设计仓库管理控制系统的一个问题及其相关的驱动因素，并从我们的模式语言中给出一个模式来解决该问题及驱动因素，同时讨论该模式怎样在系统的软件架构中实现。对于那些备选模式，我们主要介绍这些模式是什么以及我们为什么没有选择它们。

例如，在第二部分中给出的模式序列为仓库管理流程控制系统设计出的产品线架构，如图3-4所示。

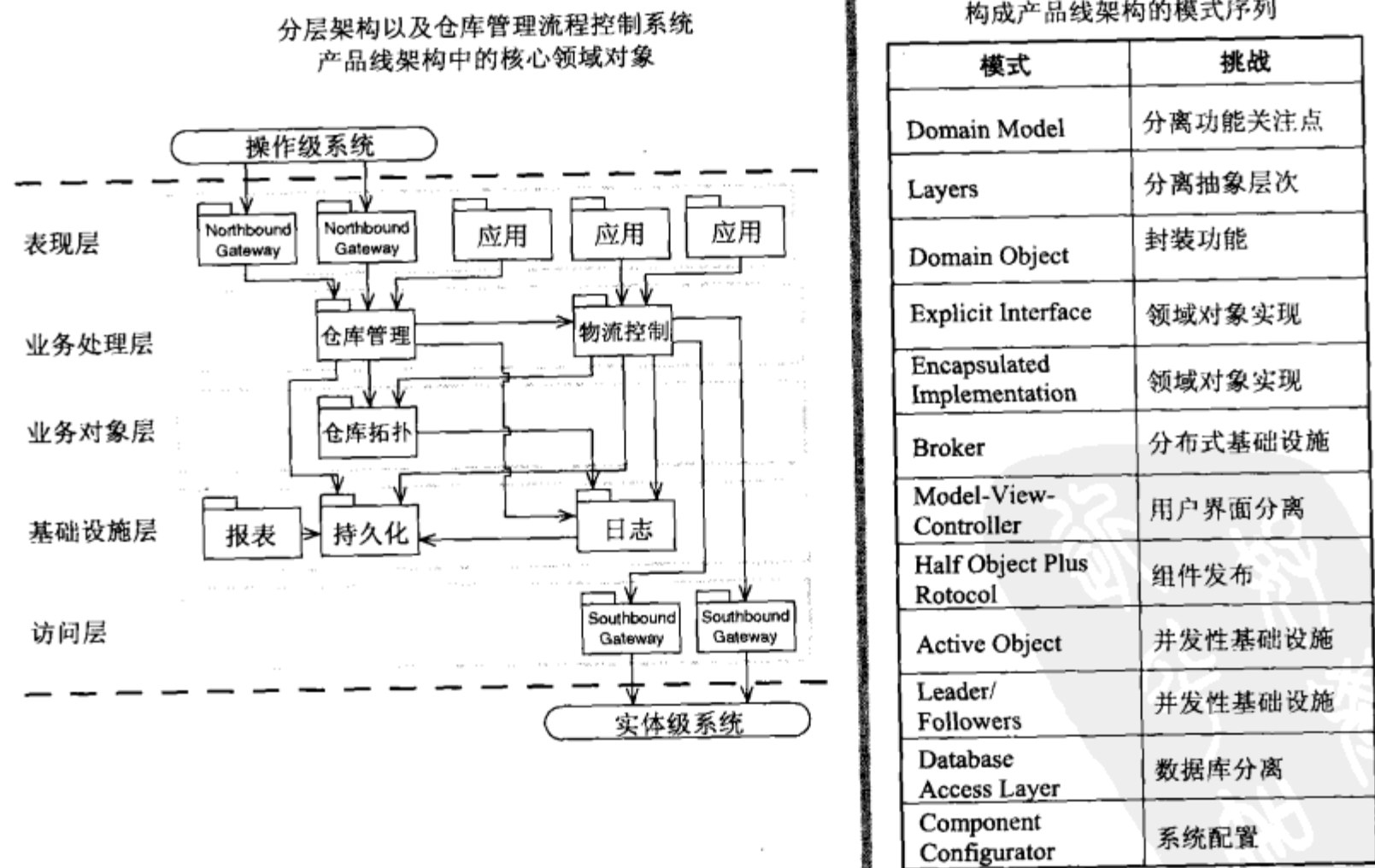


图 3-4

第二部分中给出的另一个模式序列有助于创建一个灵活高效的通信中间件产品线架构，这也是几种Broker (137) 模式实现的基础，包括Component Integrated ACE ORB (CIAO) [WSG+03]、The ACE ORB (TAO) [SNG+02] 和ZEN [KSK04]。

尽管仓库管理流程控制系统和许多其他分布式系统的架构有很多相同的特征，但它并不是一个对所有分布式系统都通用的参考架构，因为具体的需求和条件决定了所开发系统的软件架构，不同的需求和限制会要求不同的架构。不过它很形象地展示了我们运用模式语言开发分布式系统的过程，以及运用模式到产品中时需要考虑的各种因素。当在自己的环境中使用模式语言时，你需要问你的架构师、开发人员以及系统工程师类似的问题，衡量设计中的类似问题，而这些答案的获取可以参考我们的模式语言。

Part 2

第二部分

模式故事

不要让自己的行动过于胆怯和拘谨，所有的生活都是一种经历，而且经历的越多就能做得越好。

——Ralph Waldo Emerson

本书的第二部分讲述了一个模式故事：我们描述了如何应用分布式计算的模式语言去设计一个现实世界中的仓库管理流程控制系统。这部分主要关注于软件系统的3大领域：基线架构、通信中间件和仓库拓扑结构表示。

使用一个现实世界中的例子，我们阐述了怎样在模式的帮助下系统化地设计一个软件的架构。我们一步一步地展现了该架构的全貌：从基本的基线架构入手，进而审视系统内部的通信中间件，最后详细描述了表示仓库存储拓扑结构的子系统。

然而本部分的目标不仅仅为了给出一个使用分布式系统模式语言的示例，我们还希望证明模式和模式语言是开发软件架构的强有力工具，它能帮助我们更加全面深入地设计和思考，进而保证由此产生的软件系统能够满足功能需求、可操作性及开发质量。

第二部分包括以下章节。

- 第4章，仓库管理流程控制，简要介绍了仓库管理流程控制系统的领域和背景。
- 第5章，基线架构，从架构视角描述了仓库管理控制系统：划分为子系统、子系统之间的关系和交互，以及指导改进该基线架构的关键设计原则。
- 第6章，通信中间件，提供了仓库管理流程控制系统中通信基础设施的核心设计，它是对CORBA组件模型引用架构的具体实现[OMG02][OMG04a]。
- 第7章，仓库拓扑，对仓库的物理存储结构进行了概述。
- 第8章，模式故事背后的故事，这是第二部分的结束，同时对其细节、概念要点和模式语言中概念间的关系作了总结和反思。

使用现实世界中的例子来描述我们的分布式系统模式语言有一些优点和缺点。这个例子的一个突出优点是它真实发生了——因此它反映并浓缩了在开发现实的仓库流程管理系统中所进行的具体讨论和所作出的设计决定。

记录真实案例的另一个相应结果就是，其设计中的很多因素及考虑与开发的特定系统的特点息息相关，在所属领域内不一定具有普遍性，其中的一个因素就是编程语言的选择。我们的仓库管理流程控制系统的某些部分是使用C++编写的，而其他部分则是使用Java来编写的。虽然后续章节中讨论的模式序列大都与具体语言无关，但是仍有部分因素会因为选择不同的语言而不同，此外，还与标准库及框架中的可用特性有关。我们会在相关章节中适当的地方讨论这些区别。

尽管如此，系统中的特定因素并不会影响我们在这个模式故事中想要传达的关键信息：模式是帮助创建可持续的、高质量的软件架构并实现这些架构的重要工具。

仓库管理系统（WMS）是供应链的关键部分，它可提供定向的存货周转、理性的拣货指令、自动整合以及交叉收货以便最大限度地利用有用的仓储空间。系统还能基于储箱使用状况的实时信息指导和优化货物储存。部署一个仓库管理系统（WMS）意味着你不再需要依赖人的经验，因为系统本身就有这样的能力。

——维基百科

本章介绍了仓库管理流程控制系统的关键概念和需求。我们概述了系统的功能责任、运营需求以及开发中的考虑，此外，我们还阐述了如何将仓库管理流程控制系统与其他软件系统或环境集成起来。

4.1 系统范畴

仓库管理流程控制系统提供了物流支持以管理物品和资产在仓储场所内外的流动，其用户包括快递公司，如UPS、FedEx、DHL等，以及大型贸易和制造公司，如沃尔玛、宝马等。为了理解仓库管理流程控制系统的职责，以及影响其软件架构的关键因素，我们先定义其具体的范畴及其与周边系统间的关系。

仓库管理流程控制系统属于工业自动化系统大类，可以更进一步被分为以下3层，它们共同组成所谓的自动化金字塔。

- 顶层是操作层。用于管理端到端工业业务流程如企业资源规划（ERP）、制造执行（ME）以及供应链流程管理系统（SCPM）的系统有很多。SAP就是这类系统的典范，它可用于上述全部的三种活动。在我们仓库管理的环境中，操作层系统负责计划、安排并监督仓库内所有业务层操作的流程。
- 中间层是流程控制层。这一层的系统负责及时准确地执行所有操作层计划安排好的活动。仓库管理领域包括库存管理、订单管理、收货和发货管理、存储和运输工具管理等管理性任务，以及具体的运输订货这类操作性任务。我们的模式故事要讨论的主题正是位于此层。
- 底层是实体层。流程控制层的系统需要使用专业设备和网络元素以执行现实世界中的具体操作。对应于仓库管理系统，它们是表示和控制底层自动化硬件的系统，具体表现为传送带、吊车、或者人工操作的运输工具上用于人机交互的设备，例如叉车。

图4-1概括了自动化金字塔的3层模型。

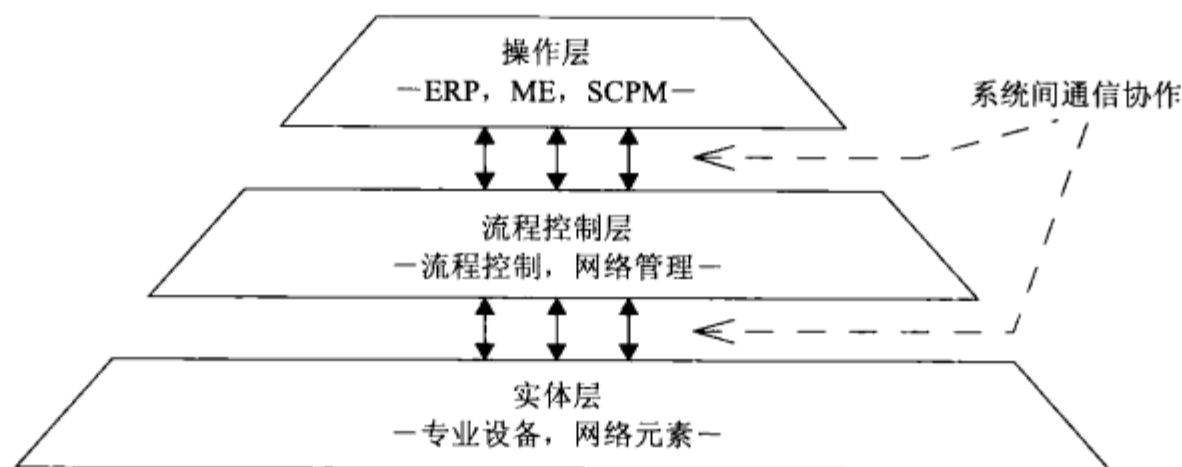


图 4-1

自动化金字塔还提出了流程控制系统的两个基础性的系统级需求。

- 流程控制系统位于金字塔的中间，它接受操作层的命令并向其汇报执行命令的所有进展，同时控制和监督底层实体层对某一特定命令的执行。最后，所有自动化处理都是端到端的，从操作层开始，在操作层结束，同时需要流程控制层和实体层参与整个控制流程。不过，因为工业自动化领域中不同的软件系统通常由不同的供应商来提供，所以要有适当的应用集成机制来完成多系统间端到端的无缝操作。
- 自动化金字塔所描绘的整个IT基础设施就其本质而言是分布式的，因此所有软件系统的架构都必须考虑到其协作系统的远程特性，且只能通过某种形式的IPC访问。

4.2 仓库管理流程控制

正如我们在上节指出的，仓库管理流程控制系统位于自动化金字塔的中间层。它一般负责执行和管理仓库中的管理型和操作性的任务，包括与操作层和实体层的系统通信。下面简要地列举了其核心功能——从而形成了系统的基础领域模型（Domain Model）。请注意我们不会列出仓库管理流程控制系统的全部职责，也不会描述所列职责的全部细节。我们的目的是让读者对仓库管理流程控制系统有一个大致的印象，从而足以理解模式故事的后续章节中采用相关设计决定和选择模式的理由。该系统的相关职责有如下这些。

- **库存管理。**对每种类型的物品，仓库管理流程控制系统都要维护相关的主数据，如物品名称、描述以及可用库存。对每一件物品，系统需要维护正确高效的订单管理所必需的数据，如最迟销售日期、当前存储时间以及最重要的——物品在仓库的存放位置。
- **订单管理。**仓库管理流程控制系统从操作层的系统接收要执行的不同类型的订单：特定顾客的发运订单，制造商或产品线的补货订单，收货部门的订单收货，以及将来的收货和发货通知。

对订单来说，系统必须首先检查所订购类型物品是否有足够的库存，如果有，下一步要决定在仓库的取货位置。这通常会基于系统维护的每种物品的信息来决定，如最迟销售日期。最后，订单管理单元生成特定的运输指令，从仓库中取出每件物品并送到指定的发货目的地以便进一步

的处理。每个订单的执行进度和状态都会反馈给自动化金字塔操作层中适当的系统。对于发货和收货通知,订单管理单元装备好相应的运输指令,如预留合适数量的物理存储位置以及运输工具,并安排它们在指定时间执行。

- **发货。**从仓库中拿到的物品必须集中起来准备发货,包括质量和数量检查、更新待发货物品的所有主数据和个体数据、打包以及打印包装单。其中一个特殊的任务是拣货:从包含有更多物品的箱子或容器中人工或自动选择一定数量的物品。这包括从仓库中选择特定的容器和箱子并将其运到拣货地点、更新相关物品的主数据和个体数据,以及将箱子或容器中剩下的物品送回仓库中的正确位置。
- **收货。**货物到达仓库之后就得开始准备存储,包括拆包、质量和数量检查,以及输入或更新所收物品的主数据和个体数据。一旦准备好,则生成运输指令将物品存储到仓库。
- **物流控制。**订单管理单元创建的运输一定数量物品的指令仅需要指明目标库存、存储目的地、盛装着物品的运输单位以及物品本身的相关信息。然而,将运输单位从指定位置移动到目的地可能包含很多步,每一步可能由不同的运输工具执行。例如,一排箱子可以用叉车从仓库门口取出并送到传送箱中,然后由吊车将其存放到货架上。将运输指令分解为不同步骤,为每一步指定合适的运输工具并监督所有步骤的执行是物流控制单元的一项职责,优化仓库内的整体物流并提高吞吐量是其另一职责。物流控制单元发送具体的运输指令给相应的自动化硬件,并接收响应的状态信息等。执行运输指令过程中的所有进展都被发送回订单管理单元。
- **拓扑管理。**仓库流程控制系统还要负责管理仓库的拓扑结构,并为订单管理和物流提供相应的拓扑表现形式。仓库中的所有的存储容器,如不同类型的箱子和货架,以及可用运输工具,如叉车、传送带以及吊车都被安排在仓库的布局结构中以确保适当而有效的仓库操作。例如,货架按过道和过道的两侧来安排,每条过道与一个或多个吊车以及传送箱连在一起,这样,吊车就可以从传送箱中拿起运输单元。仓库中的库存也是按照各种存储组织标准划分成不同区域,如存储有害物品或存储有特定温度要求的物品等。

除了以上的功能需求,仓库管理流程控制系统还必须支持一些操作性和开发性的需求。同样,为了简洁起见,我们仅关注一些相关性较强的操作性及开发性需求,以便于读者更好地理解下面章节中的模式示例。

- **分布性。**仓库管理流程控制系统天生就是分布式的。因此它的功能必须可以被许多不同的分布式客户端访问,如拣货站的PC以及叉车上的移动客户端。
- **性能。**尽管仓库管理流程控制系统不是一个“绝对的”实时系统——那种所有操作必须满足定义好的时限要求的系统,但性能仍然与业务息息相关。对系统有整体的吞吐量要求,因此系统必须确保所有的运输指令能够被及时而有效地执行,不会出现明显的中断或者时断时续的情况。
- **可伸缩性。**不同的仓库其大小可能会有很大的不同,因此仓库管理流程控制系统必须能既支持只有几千个箱子的小仓库,又要支持超过一百万个箱子的大仓库。另外,仓库所需的功能也可能大不相同,例如,根据自动化金字塔中操作层和实体层的相关系统的能

力不同,仓库管理流程控制系统提供了或多或少的管理性功能或操作性功能。最后,参与到仓库管理流程控制中的设备数目也可能相差很大:小型安装可能只需要几十个计算机设备,而大型安装可能需要几千个。

- **可用性。**许多仓库操作采用三班倒的24/7模式工作,因此可用性是仓库管理流程控制系统对业务案例支持的关键因素。任何故障中断都会影响供应链、干扰其他系统和人的状态以及操作,最终会导致业务和金钱的损失。因此,对于一般的工业自动化系统,特别是流程控制系统,通常需要保证至少99.999%的可用性——一年最多5分钟的故障时间!
- **持久化。**仓库管理流程控制系统维护的很多状态,如仓库拓扑、可用库存以及正在处理的所有订单,都必须永久保存。系统能始终依赖于一致的最新数据是非常重要的,不管是为了保留预定信息,还是为了系统在有意或意外关机后的重新启动。
- **可移植性。**系统必须能够运行于多个硬件和操作系统平台上。Windows通常是用户设备的主要选择,而UNIX或Linux主要用于核心功能服务器。类似地,系统还必须能使用不同的数据库,如Oracle和SQL Server。
- **动态配置。**仓库管理流程控制系统非常需要运行时配置(重新配置)以及部署(重新部署)。例如,仓库的容量可能临时扩展以应付季节性的高峰期。还有,根据仓库的存放内容,存取物品的策略也可能不同。然而,严格的业务可用性限制要求系统不能通过关机来(重新)配置或(重新)部署。
- **人机交互。**用户通过各种各样的用户界面与仓库管理流程控制系统通信,如基于窗体的界面(通过键盘和扫描器、有几个按键的手持终端)和成熟的图形用户界面(通过鼠标、键盘、触摸屏等)。
- **组件集成。**仓库管理流程控制系统集成了大量第三方有用或必要的产品(如数据库)和已存在的遗留系统(如为访问自动化金字塔中的实体层而用的系统)。
- **普遍性。**仓库管理流程控制系统的商业意图是提供领域内通用的解决方案,其架构和实现应当可以配置并能适应现实仓库的特定需要。例如,可伸缩性需求强调仓库大小的区别,可移植性需求列出应当支持的几种操作系统和数据库管理系统,而人机接口需求概述了要提供的一系列不同用户界面类型。根据顾客需求的不同,具体的仓库管理流程控制系统的实现在功能范畴上也有所不同。例如,一些顾客不需要库存和订单管理的功能,因为自动化金字塔中的操作层已经实现了此功能。另外,他们或许会使用其他供应商提供的具有物流控制以外的其他功能的流程控制系统。

总之,仓库管理流程控制系统必须满足很多富有挑战性的需求,它们可能是操作上的、开发上以及功能上的。系统的软件架构应当平衡各种需求,唯有如此方能恰当地满足特定仓库的需要。

每一个艺术家的思维背后，总有某种结构上的模式或者典范。

——G.K. Chesterton, 《布朗神父探案集》系列开头的独立引语

本章介绍了这个模式故事的开头：我们的仓库管理流程控制系统基线架构的规范。我们简单描述了怎样运用模式来划分系统的核心域和基础设施功能，解决分布式及并发性问题，以及为用户及其他应用程序访问、集成其功能提供支持。其结果就是产品线架构的基础：一个结构化的主干，用于获取对仓库管理流程控制系统中所有配置通用的高层级因素，同时提供基础设施和架构机制，以定义和处理特定系统实例中的变化。

5.1 架构环境

第4章中指出了仓库管理流程控制系统必须提供一系列集成有效的管理性和操作性的领域功能。根据所集成的IT环境的性能的不同，在不同的系统实例中这些功能会表现出不同的特性，甚至提供的功能集合也会有所不同。另外，实现这些领域功能则进一步要求合适的基础设施，如进程间通信、持久化以及日志。最后，系统的功能还必须能容易地被用户和自动化金字塔操作层中的系统所访问，同时仓库管理流程控制系统本身也需要能合理地访问它管理的全部实体层系统。

为了满足这些多样而又存在潜在冲突的需求集，我们需要一种方法来定义基线架构，并能够对各种冲突作出反应以平衡多样性的需求。产品线[Bosch00][ClNo01]就是其中的一种解决方案，它在软件设计层级通过采用产品线架构来实现。产品线架构是一个产品线上所有预想的成员的公共基础，它定义了对所有的系统实例都通用的结构和行为因素，还指出了哪些结构和行为会变化，从而将不变因素和可变因素明确地区分开，另外，它还支持以一种合理并可控的方式来处理可变因素[Bus03]。

设计一个高质量的产品线架构并不简单。它需要我们运用所有级别的抽象粘度和从所有角度作出深思熟虑的设计决定，有意识地应用已知的架构原则——从分离关注点，到松耦合，再到严格封装等。这意味着应该在产品线架构的基线层级对系统的主要职责进行松耦合地适当划分和模块化，提供基础设施以应对需求的多样性，并支持和预想的IT环境进行集成。

5.2 划分大泥球

实现可持续的产品线架构的基础是要清楚地划分和封装系统的不同关注点，包括功能性的关注点和基础设施相关的关注点。否则，这些不同关注点的实现会纠缠在一起，从而破坏系统的松耦合性，导致各组件的独立开发、产品线特定实例的配置，以及计算机网络中的部署都更加复杂。将不同关注点分离清楚还有另一个原因：系统不同因素的变化率是不同的。例如，系统的用户界面通常比核心功能变化得更快，而核心功能又比数据库定义变化得更快。修改应当只影响需要变化的部分，而不影响其他部分——任何涟漪效应都应当避免。

我们怎样才能将系统的功能组织成内聚的组件，以便每个组件都能独立地开发和修改呢？

将系统分成多个交互的Layers (108)，每层代表某一特定的职责或相关联的问题，并包含解决该问题的全部功能。

对于仓库管理流程控制系统来说，我们可以将其分为不同的5层。

- **表现层**。该层包含自动化金字塔中操作层系统使用的接口——所谓的“Northbound Gateway”^①，以及直接访问系统功能的用户级应用，如拣货和仓库拓扑管理。
- **业务处理层**。该层提供系统必须支持的管理性和操作性功能，如货物管理、订单管理、发货、收货以及物流控制。
- **业务对象层**。该层包含了表示领域相关的物理和逻辑实体的对象，业务处理层的功能就是在此基础上实现的。这一层的主要功能是维护和提供对仓库拓扑的访问。
- **基础设施层**。该层提供领域无关的基础设施功能，如持久化和日志功能，它是实现业务对象层和业务处理层必需的功能。
- **访问层**。该层为驻留在自动化金字塔实体层的系统提供接口，又称为“Southbound Gateway”^②。

Layers模式实现了对我们的仓库管理流程控制系统中不同关注点的严格区分。特别是它有助于我们将功能的“大泥球”划分成切实可行的抽象层次，每一层包含具有公共稳定性的元素，从而可以独立地开发和修改而不会影响其他部分产生意外后果。

5.3 层次分解

分层是为仓库管理流程控制系统提供产品线架构的重要步骤，但是仅仅分为不同层次对真正的模块化软件开发来说粒度太粗了，因为它们仅仅将不同抽象层次的功能关注点区分开，而没有将同一抽象层次不同的功能关注点区分开。例如，在我们系统的业务处理层中，仓库管理和物流控制的关注点是完全不同的，它们的耦合度非常低，但是我们仍有可能在开发过程中将其交织在一起。

① 有的地方译为“北向网关”或者“北向接口”，表示向高层提供服务的接口。——译者注

② 有的地方译为“南向网关”或者“南向接口”，表示向低层提供服务的接口。——译者注

我们怎样将Layers (108) 架构提炼成更小的、严格划分的模块部件，并为每个部件定义一个清晰的职责范畴呢？

为Layers设计中每个完备的、内聚的、功能相关的职责提供一个Domain Object (121)，从而严格划分、封装并模块化同一抽象层次上的不同的功能职责。

在系统的表现层，我们可以区分必须提供的不同Northbound Gateway和客户端应用。在业务处理层，我们可以将仓库管理功能和物流控制功能分开，仓库管理功能包含所有的管理性任务，而物流控制功能控制实体层的系统。在基础设施层，每一个不同的功能，如日志、报表和持久化也可以分开，而访问层可以为每一个支持的实体层系统^①提供一个单独的Southbound Gateway。图5-1描述了该分层结构。

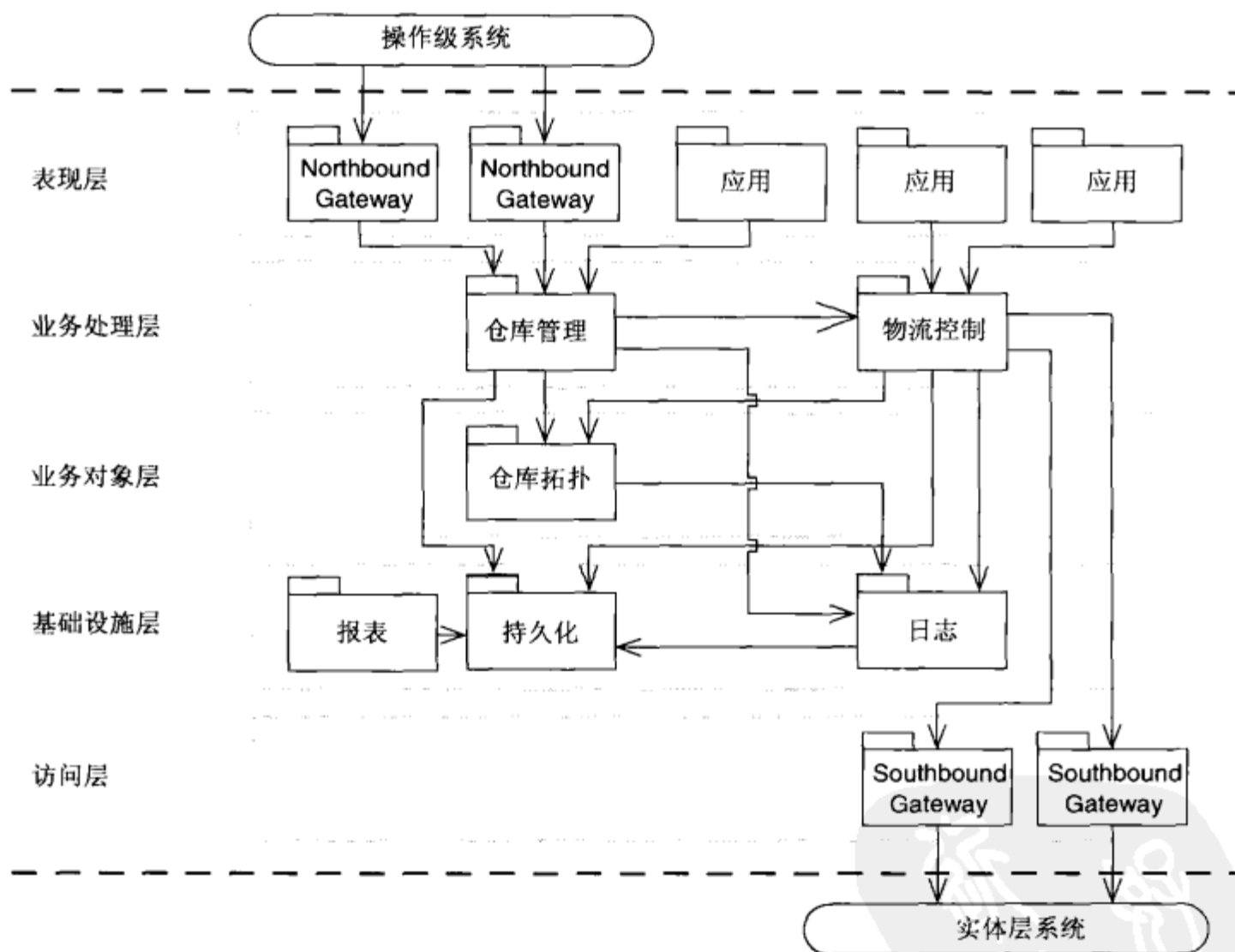


图 5-1

对模块化软件开发和产品线架构的设计来说，使用领域对象（Domain Object）划分应用功能是很理想的。领域对象提供了合适的粒度级别、关注点分离以及内聚性，以便于每个领域对象的

① 图5-1只画出了在我们的模式故事中扮演角色的功能。在现实世界的仓库管理流程控制系统中，业务处理、业务对象以及基础设施层包含更多的功能，如警告管理、监控以及安全等。此外，根据需要，我们用UML标记的包符号来表示一个领域对象。包符号允许我们表示的领域对象可以像服务一样包含多于一个的组件或类。

独立开发和改进。如果我们要对不同的系统实例做功能配置，领域对象也是一个恰当的功能配置单元。目前几种成熟的技术可用于实现领域对象，包括细粒度的面向对象框架、面向组件环境和面向服务的基础设施。

用于处理粗粒度的业务或基础设施功能的大型领域对象可以由较小的领域对象组成，从而适当地模块化其组成部分。我们的仓库管理流程控制系统中有两个这样的领域对象，分别是仓库管理和物流控制。仓库管理领域对象基本上由几个更小的领域对象组成，每个对象实现仓库管理的一个职责，如第4章所述。物流控制领域对象则分为不同层次。一个全局性的路由（route）^①领域对象负责将运输指令分解为不同步骤，并为每一步分配相应的运输设备。为了执行运输指令中的每一步，全局性的路由对象使用并控制一套领域对象来处理步骤内特定运输设备的本地路由。图5-2描述了这种划分结构。

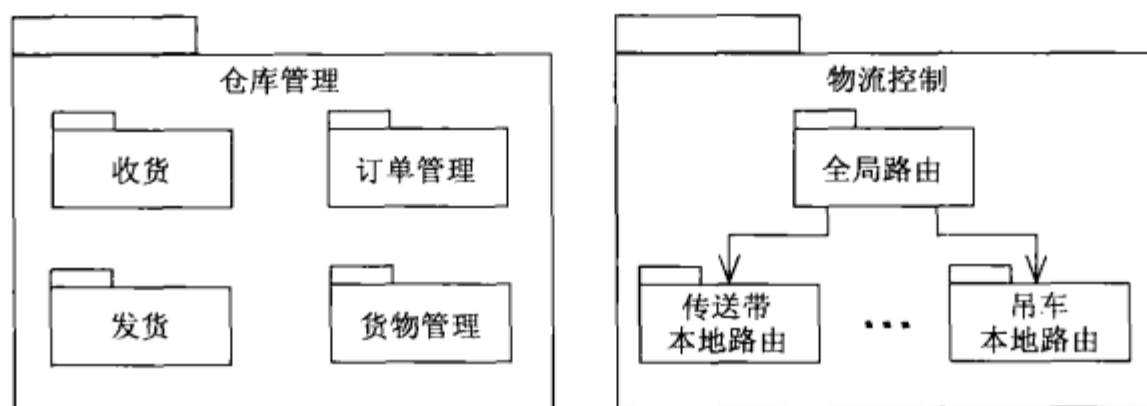


图 5-2

仓库拓扑领域对象的分解将在第7章仓库拓扑中描述。

5.4 访问领域对象功能

将仓库管理流程控制系统划分为包含领域对象的不同层，因而为模块化软件开发提供了一个可持续发展的基础，实现了开发中的松耦合和系统功能在计算机网络中的简单部署。然而，如果分析图5-1，我们会发现尽管已经清晰地分离了不同职责，领域对象之间仍然联系紧密：每个领域对象可直接访问它所使用领域对象的实现。直接访问增加了层与层之间以及领域对象之间的耦合，因为改变任意领域对象的实现都会影响到所有使用它的领域对象和层次。

我们怎样才能确保Domain Object (121) 不会依赖其他Domain Object的实现呢？

将每个领域对象分为Explicit Interface (163) 和相应的Encapsulated Implementation (181)，从而将对象公开的约定与其实现分离开。

不管领域对象和它的客户端在系统中是处于同一层，还是不同层，限定领域对象的客户端只能通过显式接口（Explicit Interface）来访问它的功能。领域对象的显式接口公布可能的客户端请

^① 这里是表示物流中的运输路线和本地传送方式，为了保持前后术语一致性，本书借用计算机用语“路由”将二者统一起来。——译者注

求集合，将这些请求——通过多态性——转发到相关的封装实现（Encapsulated Implementation）去执行，再将相应的结果返回给客户端。

Explicit Interface和Encapsulated Implementation将对领域对象的访问和其具体实现分离开，只要显式接口保持不变，改变领域对象的封装实现就不会影响其客户端。这种分离进一步支持模块化软件开发和产品线架构：其他领域对象及其开发团队能依赖所使用领域对象的稳定且定义清楚的约定，而不受其具体实现的影响。

图5-3简要描述了我们仓库管理流程控制系统的业务处理层是如何应用Explicit Interface和Encapsulated Implementation的。

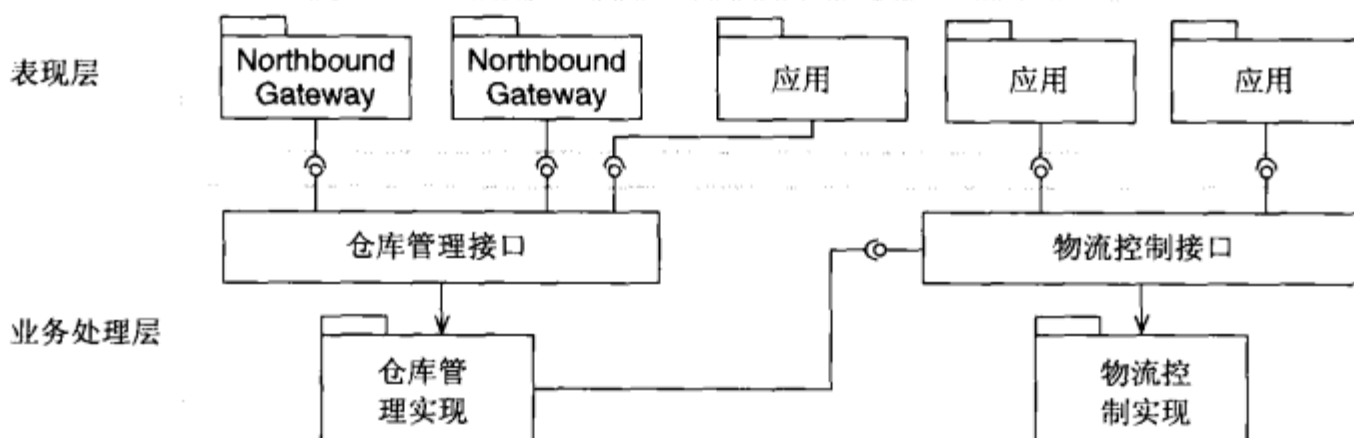


图 5-3

其他层的领域对象定义和使用对应于它们的显式接口和封装实现。

5.5 网络桥接

仓库管理流程控制系统大多部署在计算机网络上，以满足其性能、可伸缩性及可用性的需要。因此进程或机器的边界可能存在于系统中的任意两层之间，或同一层的任意两个领域对象之间。

然而，网络的引入要求我们必须面对系统基线架构的几个挑战。其中最主要的就是访问本地领域对象与访问远程领域对象的不同。在本地部署中客户端可以直接调用领域对象的操作，而在远程部署中它们必须通过网络来交互。然而，客户端不应当需要区分与其交互的领域对象是本地的还是远程的，否则它们要么依赖于特定的系统配置，要么其代码将会为了处理无数种本地和远程系统部署情况而极度膨胀。理想情况下，客户端只需要简单调用显式接口中的某个操作，而不必关心被调用的领域对象是本地还是远程的。此外，分布式系统中两个远程领域对象在交互之前，必须首先发现对方，并通过适当的在线协议建立彼此之间的网络连接，这个过程也应该与领域对象本身无关的。

我们怎样才能够保护仓库管理流程控制系统中的Domain Object (121)，使其不必直接处理网络问题并在它们之间提供位置无关的交互机制呢？

为此我们引入Broker (137) 模式以便分布式的领域对象能互相发现、访问并彼此交流，就像它们部署在一起一样。

每个网络节点的本地Broker代表系统中的领域对象进行协商并实现进程间通信的功能。远程领域对象的显式接口采用Client Proxy (139) 的方式在其客户端的地址空间实现，并处理所有与Broker之间的交互。此外，Broker还为领域对象提供注册其网络位置和所公开的显式接口的功能，并允许它们获取其他所有已注册的领域对象的显式接口，无论是本地的对象还是远程的。

Broker架构有两个重要优点：封装性以及位置无关性。封装性使得应用程序开发人员专注于提供必要的领域功能，他们不需要关心底层的网络问题。位置无关性允许客户端采用与访问同一地址空间领域对象相同的方式来访问远程领域对象，从而支持计算机网络中的灵活部署。位置无关性还有利于提升系统的可伸缩性和可用性，因为它可以通过复制和联合领域对象等方式有效地利用网络中可用的集体计算能力（Collective Computing Power）。位置无关的全部3个属性对于产品线架构环境来说非常重要，因为不同的产品线实例具有不同的功能性和操作上的需求。

图5-4描绘了在仓库管理流程控制系统的表现层和业务处理层中使用基于Broker模式的通信基础设施的情况。该图中我们还假定位于同一层中的领域对象处于同一位置。系统中不同层之间的远程交互，或同一层中不同领域对象之间的交互也相应地组织在一起。

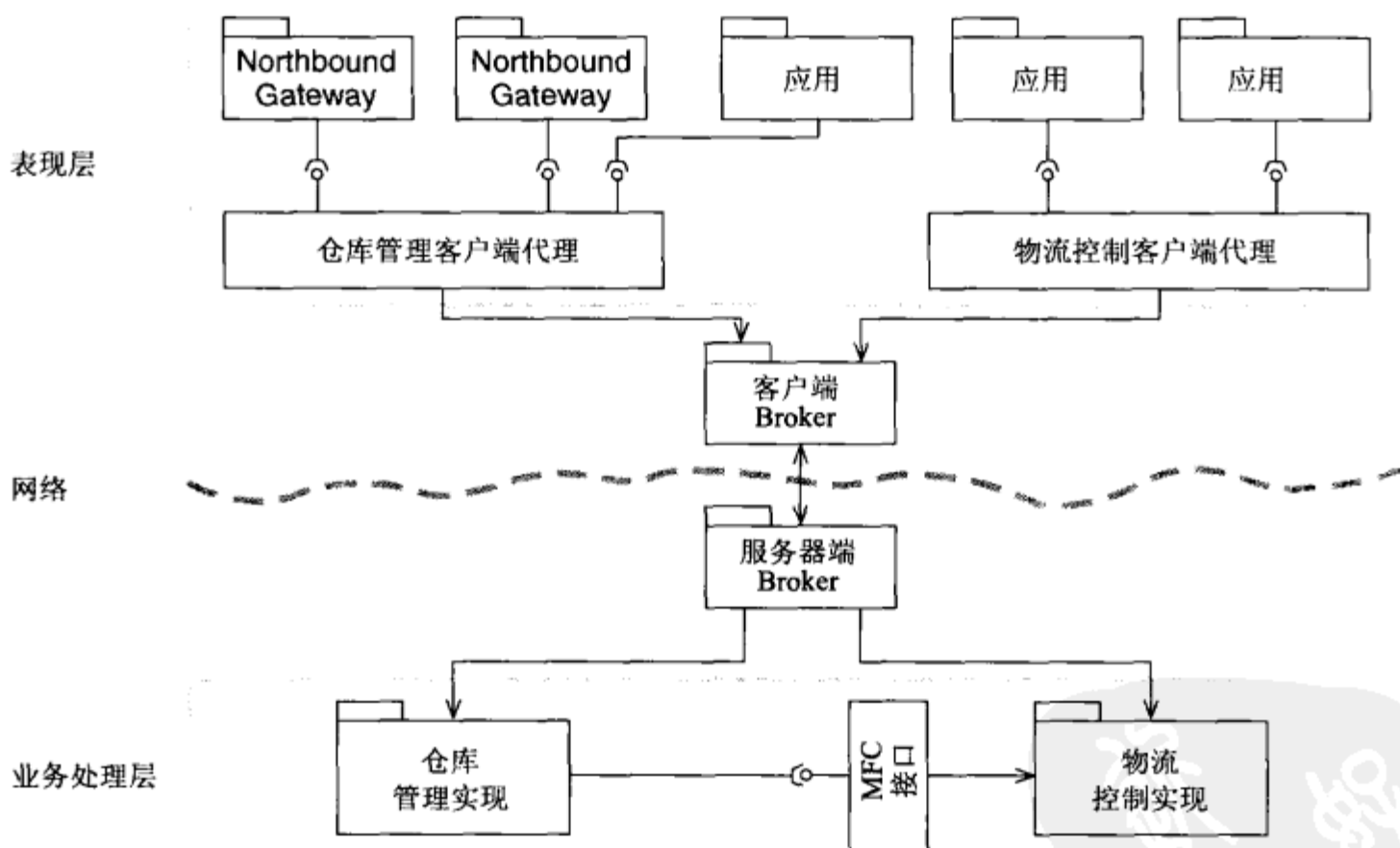


图 5-4

在仓库管理流程控制系统实际使用的Broker模式中，所有的进程间通信都是异步和面向消息的。调用客户端代理的一个方法会发出一个特定类型的异步消息——这在自动化领域被称为“请求报文”。与此类似，远程调用的结果通过“响应报文”异步返回。这里采用异步消息是因为不这样的话很难达到操作大型仓库通常所需要的性能和吞吐量方面的要求。

然而，纯粹的异步通信机制经常难以理解，更难以在应用服务的实现中正确而有效地使用。

因此客户端代理还要完成以下任务：将异步通信机制封装起来并向客户端领域对象提供它们所需要的特定通信和调用模型。这种封装有助于应用开发人员以一种更直接、易懂和方便的方式来使用异步通信机制。

在通信中间件中使用Broker方案还有另外一个对仓库管理流程控制系统的产品线开发特别适用的优点：分布式计算中间件的几种既成事实的标准都基于Broker架构，例如CORBA组件模型（CCM）、Enterprise JavaBeans（EJB），以及微软的COM和.NET技术。这使得考虑在系统通信基础设施中使用现有产品成为可能，而不必自己从头来实现。使用一个广泛接受的通信中间件标准还有利于仓库管理流程控制系统与周边系统之间的互相操作，特别是自动化金字塔中操作层与实体层的应用。

不过，在开发这个系统的时候并没有现成的中间件能满足仓库管理流程控制系统的严格要求，因此我们不得不自己设计和实现一个通信中间件。我们认识到与CORBA标准[OMG04a]保持一致有助于将来用合适的成型产品来替换。

第6章通信中间件描述了我们是如何“缩小”Broker模式来实现系统通信中间件关键元素的。

5.6 分离用户界面

按照我们目前的架构，仓库管理流程控制系统的表现层包含了提供给其他系统和访问该系统的用户级应用的网关和接口。换句话说，该层的主要职责是将功能公开给外部系统——而不管其是用户还是软件系统，并向其提供与系统当前计算状态有关的信息。表现层的领域对象既不实现任何业务逻辑，也不维护任何业务状态——这些职责都由表现层所访问的其他层实现。

当然该层所提供的信息必须是最新的。我们不能接受仓库管理流程控制系统的用户被过时的信息所误导，进而做出错误的决定。因此系统的状态变化必须立即反映在表现层的领域对象中。不过，在分层架构中严格的自上而下的访问模型使得表现层中的领域对象很难保证它们所显示的信息在任何时刻都是最新的。它们必须不断地调用底层领域对象的显式接口以获取相关信息。这样，状态的变化便无法直接反映到表现层，而是需要表现层的领域对象来轮询。因为采用轮询的方式，所以表现层保留的信息可能会过时。此外，如果没有状态变化，轮询的结果是已显示的信息，故此时不需要更新，但即便是这样，每一次轮询也会消耗系统资源和带宽。

我们怎样才能确保表现层的Domain Object (121) 始终向其客户提供最新及时的状态信息而不会破坏Layers (108) 架构的耦合原则，同时避免不必要的更新开销呢？

使用Model-View-Controller (109) 设计来使得表现层中的Domain Object与业务层中的Domain Object耦合最小，以确保它们之间的有效合作和彼此间的一致性。

业务层中的领域对象扮演模型（Model）的角色，为表现层中的应用和网关提供信息。反过来，应用和网关领域对象在向客户显示信息时扮演视图（View）的角色，在触发或者使用业务层提供的功能时则扮演控制器的角色。这三个角色之间的一套变化传播机制通知视图和控制器关于模型所维护状态的全部变化或更新信息。这样当视图和控制器收到通知后可以回调模型以获取并显示更新后的信息。

Model-View-Controller模式的主要好处在于能将状态和数据的变化及时发送给系统的用户和客户端，而只需占用极少的计算资源。它还降低了系统不同层之间的耦合——只要业务层的接口保持不变，表现层可以根据顾客的需求和UI新技术的使用独立地改进。这一点对于产品线架构环境来说尤其重要。

一种替代Model-View-Controller的方式是采用Presentation-Abstraction-Control (111) 结构，说到底，这种结构为应用中的每个不同的子系统引入特定且相互独立的用户界面。例如，仓库管理领域对象可以提供一个基于窗体的用户界面，而物流控制领域对象可以提供一个基于命令行的界面，二者间互不影响。在仓库管理流程控制系统的环境中，Presentation-Abstraction-Control模式并不能比Model-View-Controller模式提供更多的价值，而实现上却更加复杂。从应用功能中分离用户界面最简单可行的设计是采用Model-View-Controller方式，因此在我们的设计中也使用了该模式。

图5-5描述了我们仓库管理流程控制系统中的Model-View-Controller配置。视图和控制器由表现层的领域对象提供，而模型由业务处理层的领域对象表示。

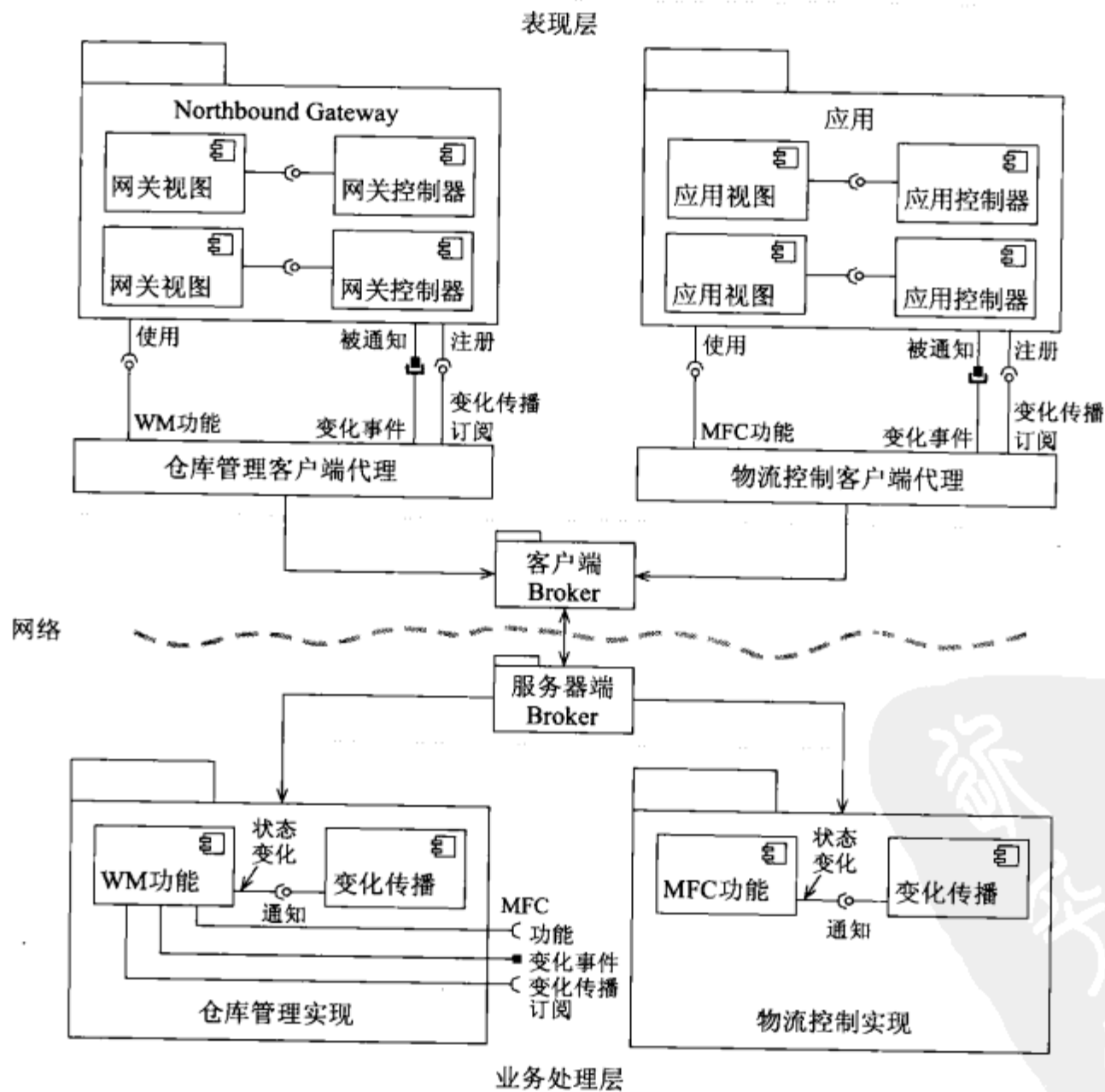


图 5-5

5.7 功能分布

图5-4（5.5节）给我们的印象是我们系统中远程领域对象的封装实现总是作为一个整体，位于单一的网络节点上，并只能通过实现在客户端地址空间的Remote Proxy (180) 来访问其显式接口。虽然这可能是最简单的分布式模型，它易于实现且容易理解，但它不能完全满足某些领域对象的性能和可伸缩性要求。

例如，如果某一个领域对象经常被许多不同地址空间的客户所访问，这种简单的分布式模型会导致额外的网络流量、延时和抖动。更糟糕的是，如果到达领域对象的请求比其处理的速度快，领域对象会变得超负荷并最终导致系统处于饱和状态。如果领域对象不维护状态，那么将其复制到所有客户端的地址空间可以解决这一问题，因为所有的复制品都可以独立执行。但是如果领域对象维护可修改的系统全局状态，比如我们流程控制系统中的仓库拓扑需要记录仓库的库存及相关传输设施的配置，那该怎么办呢？对这个问题而言，复制将不再是一个有效的方案，因为在状态变化时保持所有副本之间的一致性将会重新引入原来通过本地副本所节约的开销。

如果领域对象所维护的状态的使用会根据地点、时间和全部状态的子集而不同的话，保持副本间的一致性需要的开销就更大了。例如，在仓库管理流程控制系统采用多站点配置的情况下，就没有必要将每一个站点的全部细节复制到其他位置，因为大部分的仓库操作都只会对某个仓库产生影响。只有一小部分仓库操作会对多个站点产生影响而需要访问整个仓库拓扑，如发运和接收大量货物的订单，以及由于质量问题导致全部货物的回收等。

我们怎样才能有效访问需要维护全局状态并且客户端位于多个地址空间的Domain Object (121) 呢？

采用Half-Object plus Protocol (188) 模式来实现Domain Object，将其功能分为一套自我协调的半对象和一个位于每个客户端地址空间的半对象。

每个半对象只实现本地客户所需要的领域对象的特定功能。同样，它只维护其客户所要访问的特定数据。半对象之间采用通信协议以协调多个地址空间的活动并保持半对象状态的一致性。

就仓库拓扑领域对象来说，我们可以让每个半对象负责表示整个仓库的某一特定部分。例如，如果系统管理多站点的仓库，半对象可以位于每个站点的服务器上并只表示相应站点的特定拓扑结构。采用这种配置，绝大部分发货和收货订单业务可以通过各自站点独自进行，从而更有效率。这样即使是本地拓扑中的状态发生变化，也不需要通过网络与其他半对象协调。只有在响应某个请求的时候需要多个站点参与时，才需要引入多个半对象共同完成计算。例如，一个站点接收的订单物品总数超过该站点的能力，那么多个半对象将代表各自的客户端通过协议进行透明协商。

图5-6简要描述了在拓扑管理领域对象的设计中如何使用Half-Object plus Protocol模式。在该特定配置中一共有两个半对象参与，但是可以通过将其连接到Broker (137) 通信基础设施来增加更多的半对象。半对象之间的协议确保彼此之间的正确协作对用户来说是自动和透明的，这是通过客户端代理和Broker模式所提供的通信基础设施来实现的。

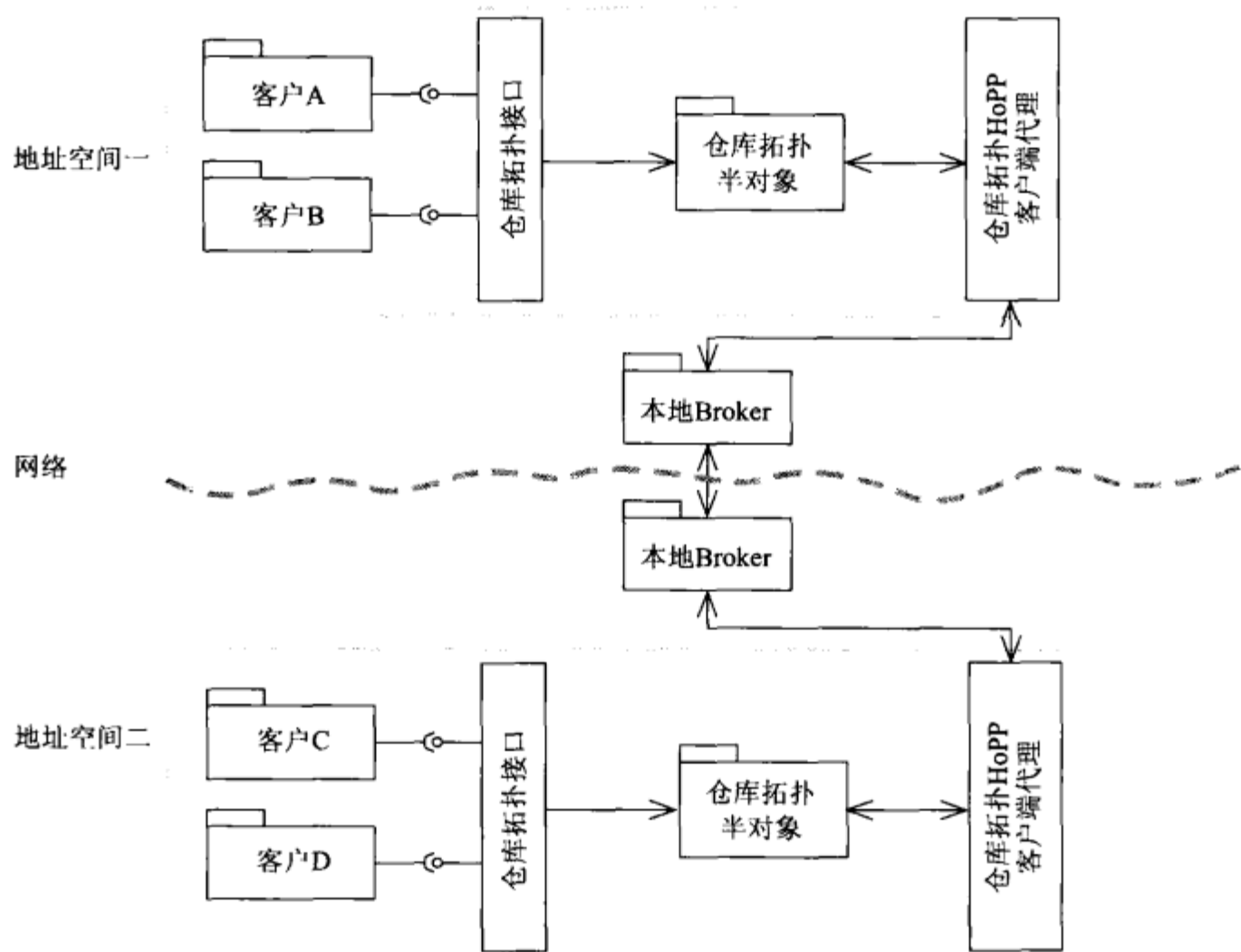


图 5-6

我们仓库管理流程控制系统的另两种领域对象也用Half-Object plus Protocol方式实现。连接仓库管理流程控制系统周边系统的Northbound Gateway和Southbound Gateway也被划分成若干个半对象，包括向外部应用提供系统功能的“标准”API的半对象和几个将某一特定应用所需的接口映射成“标准”API的半对象。与外部应用交互的半对象和这些应用处于同一地址空间中，而提供标准入口API的半对象位于仓库管理流程控制系统的地址空间中。

类似地，如果一个用户级应用设计为一个胖客户端或智能客户端，其核心功能——在客户端的Model-View-Contoller设计中扮演模型角色的部分——采用半对象方式实现并连接到相应的服务器端业务处理或业务对象层的领域对象上。这种设计能够保证所有用户级应用——不论是胖客户端、智能客户端还是瘦客户端——都能以一致的风格进行操作，并共享一致的系统状态。

Half-Object plus Protocol的设计极大地提高了我们系统的可伸缩性、性能、吞吐量、容错性以及可用性。可伸缩性的提高是因为领域对象可以利用网络上可用的分布式硬件。性能和吞吐量的提升是因为网络的使用被减到最小——绝大部分操作可以在本地的单一地址空间运行而不需要昂贵的网络开销。可用性和容错性的提高是因为半对象发生故障时不会影响其他半对象的可用性。此外，仓库拓扑领域对象的Half-Object plus Protocol设计、胖客户端和智能客户端应用以及网关有助于解决仓库大小、客户应用的复杂性以及和现有IT环境集成等不同需求。所有这些不同需求都可以由一个通用的设计来支持，这也是成功的产品线架构的关键。

5.8 支持并发的领域对象访问

采用Half-Object plus Protocol方式实现仓库管理流程控制系统中的领域对象显著提升了系统的性能、可伸缩性和吞吐量。然而，Half-Object plus Protocol只能解决客户端分布在网络上的情况。如果一个领域对象有很多并发的本地客户，那它还是会成为吞吐量的瓶颈，因为在同一时刻它只能被一个客户所访问，其他客户只能阻塞直到轮到自己，或者被告知等待，稍后再试。虽然这种行为对于一些小型部署有时还可以忍受，对那些高性能、高吞吐量和有可伸缩性要求的大型系统来说通常是不可接受的。

在这些应用中，系统的核心领域对象，如仓库拓扑等，必须随时都能访问。另外，如果核心领域对象不能立即提供服务，则并发的客户端也绝不能阻塞。同样也不能拒绝这些客户提出请求——这对高优先级的请求尤其重要，如运输设备（或某部分）的紧急停止。此外，如果系统安装于或更新到具有更多计算能力的硬件上，如多处理器，那么它应当可以直接使用这些额外资源而不需要修改系统实现。

我们怎样才能对一个共享的Domain Object (121) 提供并发访问，以便客户端可以随时提出请求而不会阻塞，并允许领域对象按任意顺序来处理这些请求从而确保高吞吐量呢？

采用Active Object (212) 实现Domain Object，从而将提出请求与执行请求在时间和空间上分开。

领域对象的封装实现运行在单独的线程池中，而其显式接口在客户端的线程中提供。这样客户端可以向显式接口发出请求而不会阻塞，在请求处理过程中可以继续执行其他任务，并在需要的时候访问执行结果。显式接口将其收到的所有请求对象化，以便领域对象的封装实现按照限制条件采用合适的顺序安排执行。

例如，在仓库拓扑领域对象的Half-Object plus Protocol (188) 设计中，每一个参与的半对象都实现为一个Active Object。因此当在多处理器机器上执行时，采用这种方式的任意Active Object可以并行处理多个请求。为了最有效地利用可用的处理能力，仓库拓扑半对象及其活动对象的任务分配和现实中的配置相对应——仓库中的那些可以彼此独立并行操作的部分，如不同的仓库建筑，会被分配给不同的活动对象或一个活动对象的不同线程，这样它们就可以在软件系统中独立并行地运行。而那些不能独立或并行运行的部分，如被同一个吊车使用的所有高架轨道，会被分配给同一个活动对象甚至同一活动对象的同一线程。因此，将现实世界中仓库拓扑结构的并行性集中在一套半对象中，并实现为活动对象，可以显著地提高领域对象的吞吐量。

图5-7阐述了仓库拓扑领域对象的Half-Object plus Protocol设计中一个特定半对象的活动对象分布^①。

^① 这里我们再次按照我们的需要对UML标记作了部分改动。UML没有提供一种方便的形式来表示一个活动组件运行在自己的控制线程中。为了尽可能地接近UML标记，我们采用类似于活动对象的方式标记活动组件，并在组件符号中使用并行线表示活动类。

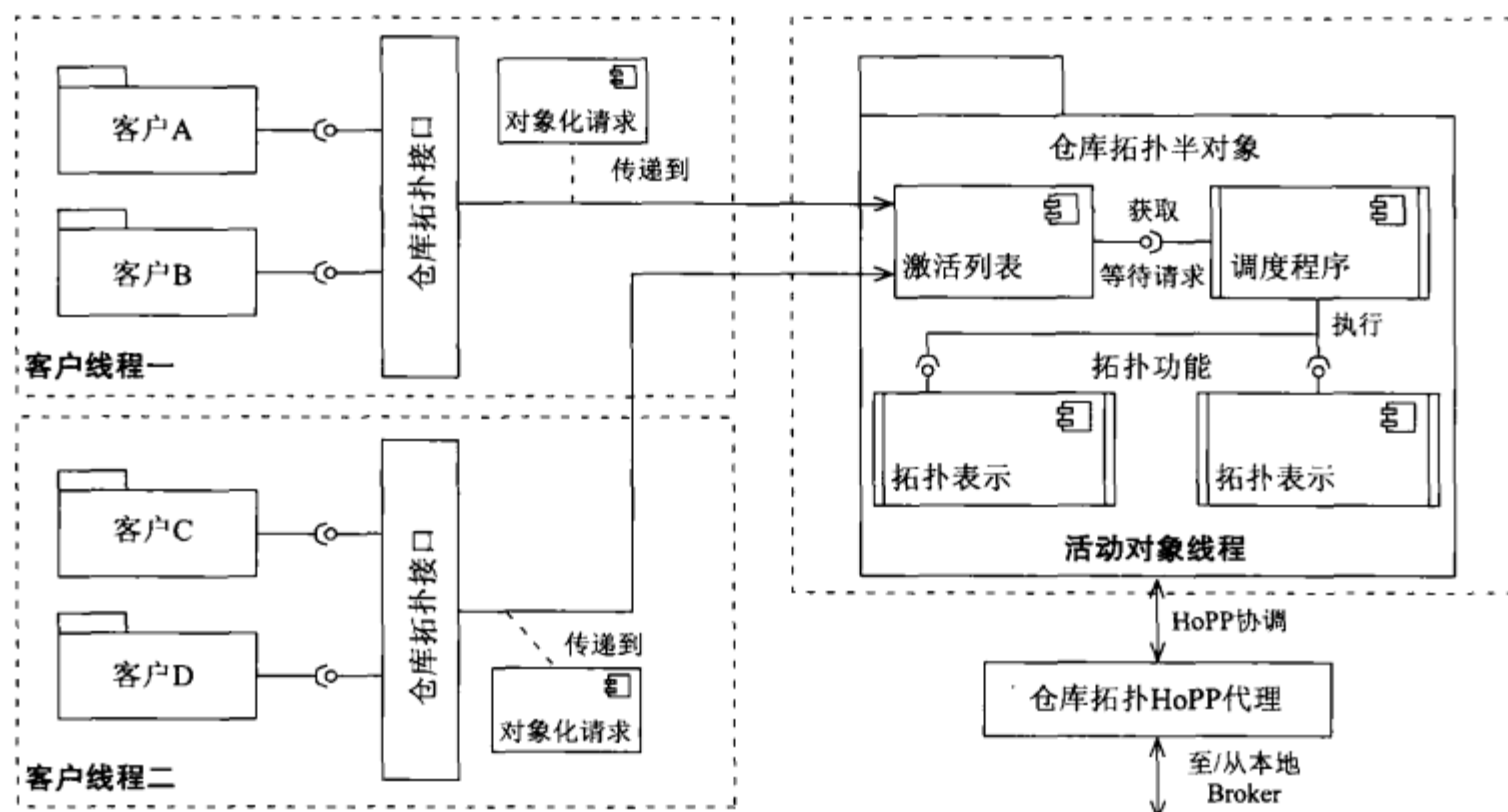


图 5-7

5.9 获得可扩展的并发性

为了支持领域功能的执行，仓库管理流程控制系统实现了一套不同的基础服务。其中的一种服务是日志——将系统领域对象产生的各种信息记录（如错误和轨迹）永久性地存储起来以备日后检查。使用日志功能应当只对系统的运行质量产生很小的影响，特别是性能——领域对象不能因为日志对象不能及时处理接收到的日志记录而导致阻塞。然而根据当前的运行状态可能会有大量事件需要记录，从而导致日志的量非常大。通过为每个网络节点提供单独的日志领域对象，在分布式的持久化存储单元中记录日志信息可以减少该问题的发生，但它对于网络中某一特定节点产生大量日志记录的情况并没有什么帮助。

我们怎样才能避免即使在处理大量日志记录的情况下日志也不会成为性能的障碍呢？

使用Leader/Followers (211) 并发模型来实现日志Domain Object (121)，它通过使用一个预先分配的线程池来避免动态的多线程开销。

到达特定网络节点日志领域对象的日志记录经过分析、处理，最终储存到分布式持久化存储单元的一个节点的本地实例中——这是通过具有自我调节功能的线程池来完成的。其中领导者线程监听日志记录的到来，其他线程作为跟随者等候，直到轮到自己来监听日志记录。当领导者线程接收到日志记录，它将自己转化成处理线程的角色，同时将一个等待的跟随者线程提升为领导者线程，然后处理并存储接收到的日志记录。因此多处理线程可以同时运行并处理大量的日志记录。一旦日志记录被存储到持久化存储单元，处理线程再将自己转换成跟随者线程并等待再次成

为领导者线程。

Leader/Followers设计的关键好处是性能方面的——对执行小的、重复性的原子行为，比如处理日志记录，它提供了一种高效的、节约资源的并发模型。如图5-8所示，一个不停旋转的线程“轮子”处理所有进入的日志记录。理想情况下，线程池的大小应当能处理最大期待负荷，这样到达日志领域对象的日志记录能立即被接收和处理。

Leader/Followers设计的一个备选方案是Half-Sync/Half-Async (209) 结构，使用一个适当大小的队列来缓存那些到达速度比处理速度更快的日志记录。然而，这个方案没有Leader/Followers直接和优雅。因此在系统设计中优先选择Leader/Followers，而不是Half-Sync/Half-Async方案。

图5-8展示了日志领域对象的Leader/Followers设计，它采用有5个线程的线程池。从日志接口接收日志记录的线程扮演领导者线程角色，两个正在将日志记录存储到持久化接口的线程是处理线程，剩下的两个线程作为跟随者线程等待成为新的领导者线程。

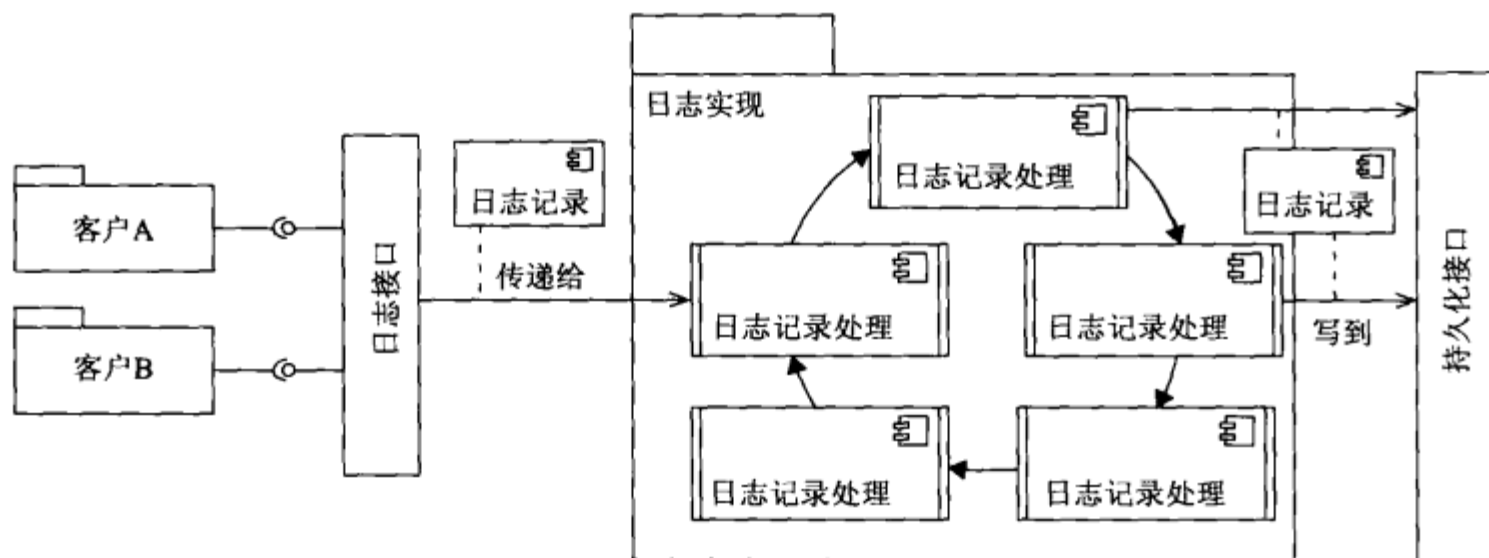


图 5-8

一个后台的报表领域对象根据存储在分布式持久化存储单元的日志记录生成各种报表。每条记录的时间戳是由分布式时间领域对象产生的，只要确保网络上互相协作的计算机之间的时钟精确同步，我们就能将存储在不同网络节点的日志记录正确排序。

5.10 将面向对象与关系型数据库连接起来

仓库管理流程控制系统所创建和维护的所有数据都必须持久化存储，而且其更新必须是事务性的，这样才能保证系统始终根据一致和最新的仓库当前状态信息来运行。因此基础设施层提供了持久化领域对象，以便其他领域对象能够向数据库中写入自身业务对象或从数据库中获取业务对象。

由于一些典型的非技术原因，数据库通常由主要的数据库提供商之一提供，这就意味着它通常采用关系范式。然而，将数据库的关系型特征直接暴露给仓库管理流程控制系统——这样一种采用行为对象来设计和实现的系统，会导致范式的不匹配。业务对象，如发货订单、仓库拓扑元

素以及日志记录, 必须在传递到持久化领域对象之前首先转换成适当的表结构。相反, 表中组织的数据必须首先转换成适当形式的业务对象, 然后才能被领域对象使用和处理。换句话说, 仓库管理流程控制系统的应用级代码被打乱并依赖于面向表的数据结构和SQL查询。如果我们需要将系统移植到另一个数据库, 情况将变得更差, 很可能新数据库提供不同的接口, 从而要求处理持久化的所有代码做出相应修改。

我们怎样才能将仓库管理流程控制系统中所使用业务对象的面向对象视图与数据库要求的关系型视图之间连接起来, 而不需要将一种视图强加到另一种之上呢?

在仓库管理流程控制系统和关系型数据库之间引入Database Access Layer (318), 从而将应用中逻辑的、领域特定的数据表现与表中数据的物理表现分离开来。

持久化领域对象的显式接口提供了系统预期的持久化存储视图, 允许采用面向对象世界中的建模方式传入和获取业务对象。持久化领域对象的封装实现提供了面向对象结构(业务对象要求的结构)和表结构(关系型数据库接口要求的结构)之间的双向映射。将持久化领域对象移植到另一个数据库所需的全部改动都被置于Database Access Layer的设计中。将数据库的不同封装在稳定的接口后面显著地提高了我们系统基线架构的产品线特征。

图5-9描述了该设计。

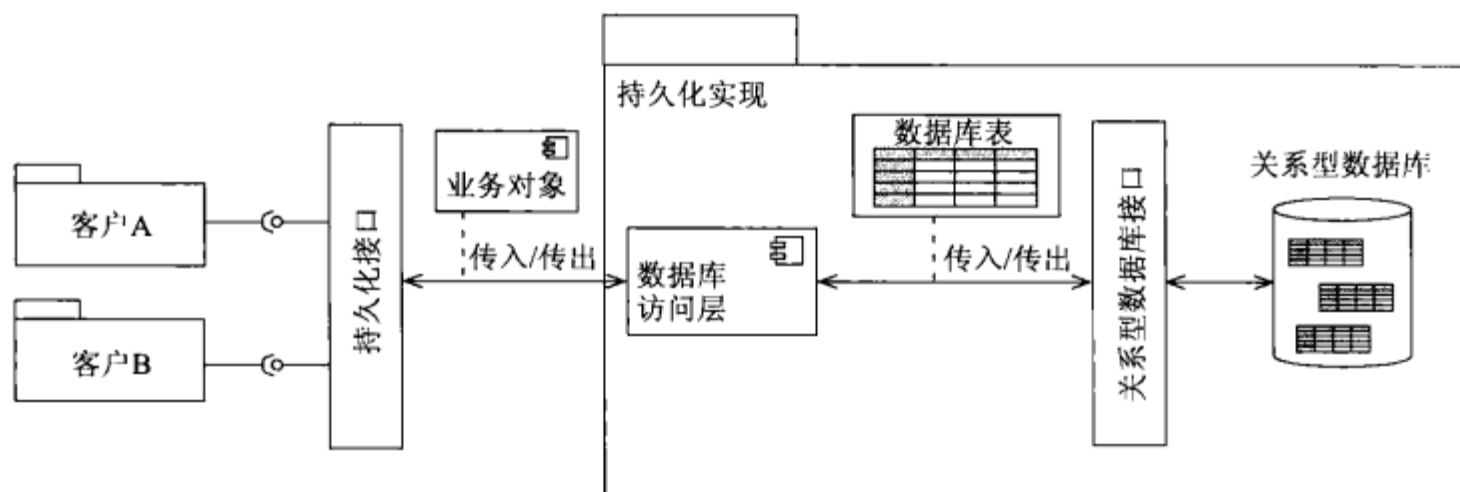


图 5-9

5.11 领域对象的运行时配置

我们仓库管理流程控制系统要求99.999%的可用率, 也就意味着一年总共只有不到5分15秒的故障停机时间。然而, 系统运行中有很多东西都会发生变化。例如, 仓库物理结构的变化会触发系统的重新配置, 以便为特定领域对象安排不同的算法, 甚至是采用不同的领域对象实现。系统运行的计算机网络变化, 如增加新的服务器, 可能需要重新部署领域对象以便更好地应用新的硬件。仓库管理流程控制系统必须能灵活响应这种重新配置和重新部署的需要, 而不会降低系统的可用性。

我们怎样才能支持对仓库管理流程控制系统进行灵活的（重新）配置和（重新）部署，而不需要关闭整个系统呢？

提供Component Configurator (289) 基础设施以支持Domain Object (121) 的动态、运行时（重新）配置和（重新）部署，而不会影响系统中与这些活动不相关的部分的可用性。

Component Configurator基础设施允许向运行中的系统中动态装载新的领域对象实现和配置，并作为系统中心控制实例来协调实现和配置的运行时切换、（重新）配置以及（重新）部署等。为了能重新配置和重新部署，领域对象的显式接口必须实现Component Configurator能访问的生命周期管理功能，以便对其进行正确有效的控制和管理，而不会导致系统整体状态的不一致。因此Component Configurator基础设施平衡了高可用性和支持系统改进的需求之间的冲突。

5.12 基线架构总结

前面几节描述了我们怎样运用本书第二部分模式语言中的模式构造仓库管理流程控制系统的基线架构。本节我们从整体上分析所应用的模式序列，并概述在仓库管理流程控制系统的基线架构设计中比较适用的一些普遍特性。

模式序列中的前两个是Layers (108) 和Domain Object (121)。这两个模式都有助于我们理解仓库管理流程控制系统，允许我们将这个“大泥球”按照抽象层次划分成能够处理的模块。这种划分是二维的，分层提供了一种基础的、水平方向的分解，将不同的关注点互相分开，例如将外观因素与业务和基础设施逻辑区分开。领域对象提供了垂直方向的分解，将每一层划分为不同的职责，例如将仓库管理功能与物流控制功能分开。运用分层和领域对象的结果是从职责和核心使用关系上将所有基线架构元素清楚地识别出来，并进行模块化的分割。这样，对应用功能的水平和垂直分解是几乎所有软件架构的基础——无论是开发什么样的应用，在一个有目的的、基于模式的软件开发过程中Layers和Domain Object通常是首先采用的模式。

接下来的两个模式，Explicit Interface (163) 和Encapsulated Implementation (181) 将领域对象的技术实现和提供的功能分离开。改变领域对象的封装实现不影响它的客户，它们可以按照一个稳定的约定来编程。这两种模式允许Layers和Domain Object的划分结果采用合适的模块化软件技术实现，如组件或服务。因而这两种模式不仅可以应用在仓库管理流程控制系统的环境中，还有助于定义其他许多必须采用模块化软件开发的应用架构。

第5个模式——Broker (137)，通过在领域对象之间定义清楚的网络界限使其可以彼此都是远程的，从而与仓库管理流程控制系统的分布式特征相符。不过Broker的引入不仅仅允许两个领域对象可以驻留在不同的地址空间或者不同的网络节点上，它实际上定义了一整套进程间通信的思想，从领域对象怎样公布它们在系统中的可用性开始，到两个或更多领域对象怎样建立彼此之间的通信渠道，再到这些领域对象怎样通过通信渠道进行通信和交互。

Broker模式所主张的分布式思想解决了许多分布式系统的远程需求，而不仅仅是我们的仓库管理流程控制系统。大多数中间件，如CORBA、.NET以及J2EE都采用Broker作为进程间通信功能的核心架构。因此我们在第6章通信中间件详细描述了Broker设计。

模式序列中的第6个模式，Model-View-Controller (109)，将仓库管理流程控制系统用户界面层的领域对象和业务处理层及业务对象层的领域对象组织起来并互相合作。按照Layers模式，高层领域对象可以调用低层领域对象的显式接口，反之则不行。这个规则引入了分层系统中的一个问题，那就是控制流并不总是从用户界面开始“流到”运行时基础设施层。例如，在事件驱动系统中控制流通常由底层产生而逐渐“爬升”到最上层。在多用户界面客户端所访问的系统中，状态变化要求所有这些客户端同时更新以正确显示信息。

Model-View-Controller模式通过控制流反转解决了上述问题，这就是好莱坞原则：“不要找我们，我们去找你” [Vlis98a]。应用数据和状态并不由视图和控制器维护，而是在模型中改变，模型负责维护这些信息并通知视图和控制器，这样视图和控制器可以翻过来根据需要调用模型的显式接口来更新自己的状态。控制流可以在不同层中交换，而不需要引入低层对高层不必要的依赖。就像目前应用的其他模式一样，Model-View-Controller并不仅仅适用于仓库管理流程控制系统，而是所有提供交互式用户界面应用的通用结构。

模式序列接下来的3个模式旨在为各种领域对象提供合适的运行质量（operational quality）。Half-Object plus Protocol (188) 引入了联盟机制（federation），以支持多个地址空间访问的、状态丰富的领域对象的性能、可伸缩性以及可用性。Active Object (212) 以及Leader/Followers (211) 模式使用定义好的并发模型，以保证单一地址空间领域对象实现的高吞吐量。所有这三种模式提供了通用的分布性和并发性模型，并可应用于仓库流程领域之外的许多系统。

Database Access Layer (318) 是一种通用的解决方案，它使得软件系统架构不必关心第三方数据库所引入的范式阻抗不匹配和实现细节。将一种数据库产品换成另一种仅需要对数据访问层内部进行的修改——尽管改动可能会很大。数据库的客户端并不受影响，这种改变不会使它们受到牵连。同时，引入Component Configurator (289) 完善了基线架构中领域对象的松耦合，新引入的基础设施在软件更新和环境改变时支持领域对象的动态灵活配置和部署。

下表总结了帮助我们构建仓库管理流程控制系统基线架构的模式序列，以及序列中的每个模式解决的设计挑战。

模 式	挑 战
Layers	根据不同抽象层次划分应用功能
Domain Object	在同一抽象层次内部划分和模块化应用功能
Explicit Interface	为Domain Object提供定义良好的访问接口
Encapsulated Implementation	提供并封装Domain Object的实现
Broker	定义通信中间件的基线架构
Model-View-Controller	将应用功能和表现及控制区分开
Half-Object plus Protocol	支持贯穿分布式边界的联盟式Domain Object
Active Object	为Domain Object必须支持的请求调度提供并发性支持
Leader/Followers	为需要大吞吐量的Domain Object提供并发性支持
Database Access Layer	将应用功能从数据库细节中解脱出来
Component Configurator	由可重用组件为应用提供动态配置功能

对模式序列的分析表明它们并不是和仓库管理领域紧密相关的。相反，它是一个非常通用的序列，可以应用于许多分布式系统的开发。这种“普遍性”表明，分布式系统在本质上共享许多特性和需求，而与具体的应用领域无关。因此，很自然本书第三部分描述的分布式计算模式语言同样支持这个模式序列。

该模式序列中的大多数模式同样有助于仓库管理流程控制系统的产品线架构设计。Layers、Domain Object、Explicit Interface、Encapsulated Implementation和Component Configurator支持适当粒度的分解和松耦合；Broker模式支持领域对象在计算机网络中的分布；Active Object和Leader/Followers支持灵活的并发性模型；Model-View-Controller以及Database Access Layer将结构上的区别封装在通用的设计和稳定的接口中，而后者对产品线架构至关重要，如果领域中的每一种潜在变化都需要做特殊处理，那为所有产品线实例共享的通用内核定义一个稳定架构——即使可能的话——将会非常困难。通过通用的设计捕捉不同的变化则避免了这一问题，从而支持产品的所有变化，因此使用同一基线架构更为简洁。

上述模式序列对产品线架构的内在支持同样是与仓库管理领域无关的。模式从总体上记录了框架、平台以及产品线设计的最佳实践[John97]，因此这些实践同样为我们的分布式计算模式语言所支持。



好的交流就如一杯黑咖啡，让人彻夜难眠。

——Anne Morrow Lindbergh

本章通过一个案例展示如何应用本书第二部分的分布式计算模式语言中的一个关键模式序列。它描述了我们的仓库管理流程控制系统中所使用的通信中间件的开发过程，同时还介绍了在仓库管理和其他领域的一些其他的分布式应用的开发情况。该中间件允许客户端调用分布式对象上的操作而不必关心对象的位置、编程语言、操作系统平台、通信协议或互连，以及相关的硬件资源。我们的通信中间件的新颖之处在于它的设计和实现是高度可配置、可伸缩、可移植的，它经过裁剪便可以适应某个特定的应用需求和网络/终端系统特性，这比自己动手编写代码或者使用传统的中间件实现要简单得多，传统的中间件的实现中往往都通过硬编码的方式固化了一套策略。

6.1 分布式系统的中间件架构

我们在仓库管理流程控制系统的具体部署过程中经常会遇到不同的硬件和软件平台。例如，客户端应用和用户界面往往部署到Windows PC上面；传感器和制动器（actuator）则通常部署在运行VxWorks的嵌入式设备上面；代表业务逻辑和基础设施功能的Domain Objects (121)则往往部署在运行Solaris或者Linux的服务器上面。不同的设备通过不同类型的网络连接在一起，比如无线或有线的LAN和WAN，同时也会用到不同的通信协议，比如TCP/IP、PROFibus、VME等。每个系统的安装又必须和遗留的或者第三方的软件集成在一起，尤其是那些位于“自动化金字塔”操作层（operational level）和实体层（entity level）的软件，它们往往用不同的语言编写，如C、C++、Java、C#等等。由此产生的异质性给系统开发和集成提出了巨大的挑战，尤其是移除软件组件或者用其他供应商的组件来代替已有组件的时候。

在分布式系统中，通信中间件位于客户端和服务端之间，比如Common Object Request Broker Architecture (CORBA) [OMG04a]和Enterprise Java Beans (EJB) [Sun03][Sun04a]。通信中间件的目标是通过为底层的——往往是异质的——网络和操作系统服务提供一个一致的视图，简化应用程序的开发和集成。而且，中间件还有助于将一些复杂的关于分布式系统基础设施的任务由应用程序的开发者转移给中间件开发者[ScSc01]，由他们来实现通用的网络编程机制，比如连接管

理、数据传输、事件和请求分类、(解)封送和并发控制等。

为了简化系统中分布式领域对象之间跨进程的通信,并为它们的实现屏蔽计算环境的异质性,系统的基线架构使用了基于Broker (137) 的通信中间件。Broker允许分布式领域对象互相发现、访问和彼此间通信,就像事先搭配好了一样;同时,在分布式系统中对其进行解耦合,这样它们才可以在异质的环境中应用不同的技术开发和集成。

图6-1展示了如何使用基于Broker的通信中间件作为系统表现层和业务处理层。该图假设同一层中的领域对象是互相搭配的。系统跨层的或者同一层中不同领域对象的交互相应地组织在一起。

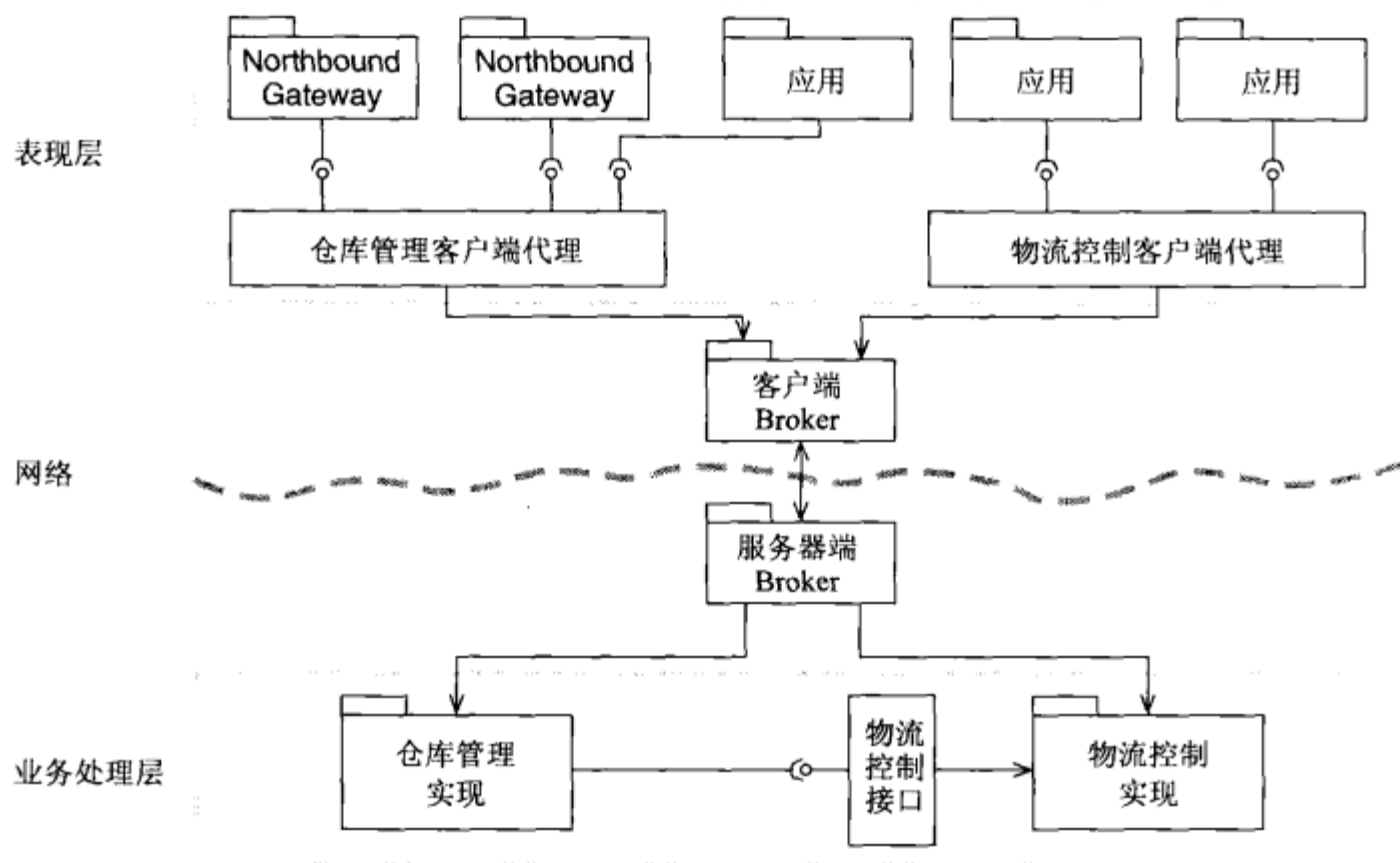


图 6-1

我们使用CORBA组件模型 (CORBA Component Model, CCM) [OMG02]来实现基于Broker的系统通信中间件。CCM通信中间件允许应用程序调用组件实例上的操作,而无需关心其位置、编程语言、宿主平台和网络协议。CCM本质上是一种语言,它是EJB的一个与平台无关的变体,同时它也支持微软的COM和.NET中的一些功能特性。CCM引用模型的核心是Broker,其中包含的元素如图6-2所示。

CCM引用模型定义了如下的关键实体。

- 客户端和组件实现运行在通信中间件上面的应用程序。
- 对象请求Broker核心^① (Object Request Broker Core, ORB核心) 负责将客户端的操作请求传递给组件实例,如果有需要还可以返回其响应。

① 为了和较早版本的CORBA规范保持兼容,OMG CCM规范使用了术语Object Request Broker, 尽管使用CCM的应用开发人员编程时往往是跟组件实例打交道。

- ORB接口将应用与ORB核心的实现细节解耦合。
- IDL桩 (Stub) 和骨架 (Skeleton) 则是客户端与服务器组件之间重要的“粘合剂”，同时它本身也是一个ORB。
- 容器和对象适配器将组件实例和ORB连接起来，它负责为管理组件生命周期属性提供一个运行时环境，分离刚到达的向组件实例的请求，分派那个实例上合适的上行方法调用。

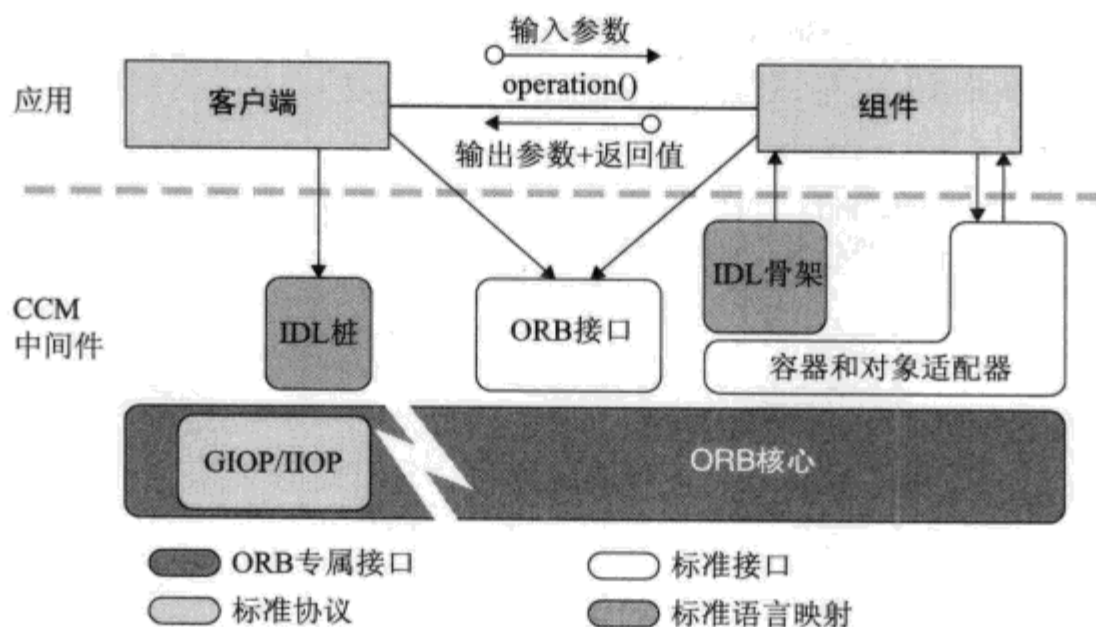


图 6-2

然而，CCM引用模型有意只是指定了ORB中必备的重要角色，而没有定义一个具体的软件架构并使之成为其他CCM实现的基础。因此，我们使用Broker模式来定义实际的组件、组件之间的关系，实现适合CCM的通信中间件所需的协作，如图6-3所示。

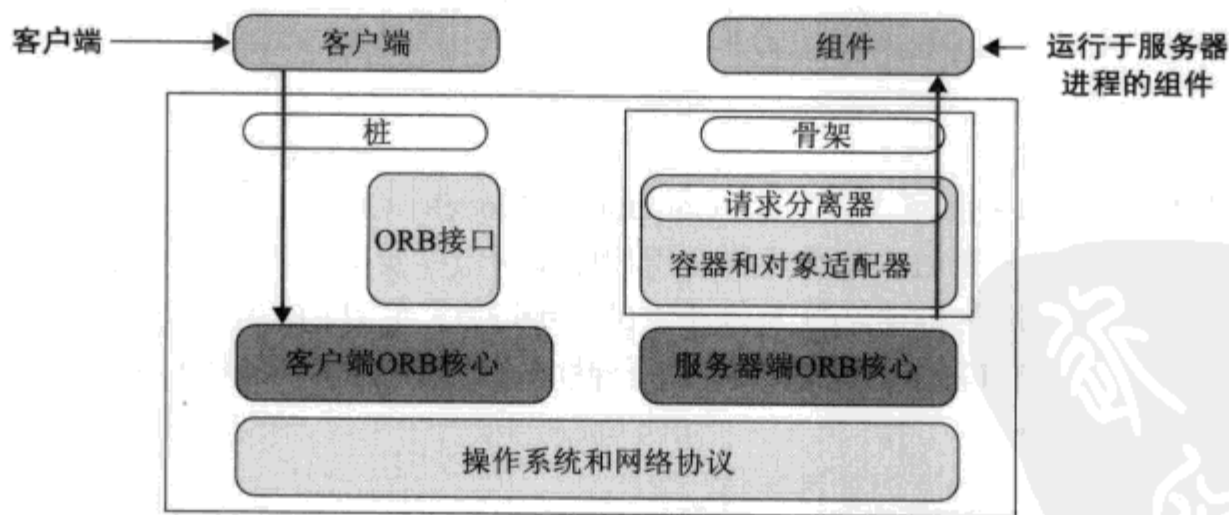


图 6-3

Broker模式中客户端和远程组件角色代表了OMG CCM引用模型中应用层次的客户端和组件。Client Proxy (139) 角色由ORB的桩实现，它提供了对远程组件服务的访问。桩也使得客户端和服务端不必关心它们的远程通信对象的位置和实现细节，以及ORB的实现细节。

客户端ORB核心扮演Broker模式中的Requestor (140) 的角色，服务器端的ORB核心则扮演

Invoker的角色。容器和对象适配器扮演Container (288) 和Object Adapter (256) 的角色。所有这些角色负责以位置透明的方式将请求从客户端传输到服务器, 同时将响应和异常从服务器传回客户端。服务器和客户端的ORB核心分别负责为服务器和客户端提供API来注册组件实例和调用实例上的方法。这些API代表CCM引用模型的ORB接口, 并且通常使用Facade (171) 模式来实现。

使用Broker模式来实现基于CCM的ORB要求解决几个设计方面的挑战。这些挑战中主要包括结构化ORB的内部设计来分离关注点; 封装底层系统功能来增强可移植性; 分离ORB核心事件; 高效而灵活地管理ORB连接; 通过并发地处理请求和使用高效的同步请求队列来增强ORB的可伸缩性, 使内部ORB机制之间可互换, 将这些机制联合形成语义兼容的策略, 动态地配置这些ORB策略。本章的剩余部分描述了我们用来应对这些挑战的模式序列。由此产生的通信中间件为我们的系统提供了Broker平台, 同时也提供了仓库管理领域的其他部分。该中间件也可以应用于其他领域的分布式系统, 包括通信、电子商务、航天、在线金融服务和电子医疗影像系统。

6.2 对中间件的内部设计进行结构化

基于CCM的通信中间件有几个重要的职责, 比如为客户端和组件提供API, 将请求从客户端路由到本地或者远程的组件实例, 并将响应返回给客户端, 初始化此类请求和响应的网络传输。由Broker (137) 定义的该架构将应用逻辑从通信中间件功能中分离出来。然而, 基于CCM的ORB本身太复杂了, 以致无法实现为一个独立完整的组件。而且它负责覆盖不同类型的功能, 这些功能可以组织成层级结构。例如, API位于应用层、组件策略管理位于容器层、请求路由和并发控制位于对象适配器和ORB核心层, 请求(再)传输位于操作系统和网络协议层。

因此, 我们需要进一步分解基于CCM的ORB架构, 以便使它符合以下要求。

- 可修改。对ORB一个部分的增强和改变应该限制在少数几个组件上面, 而不应当影响其他无关的组件。
- 稳定。外部接口应该是稳定的, 甚至有可能是由标准指定的, 比如OMG CCM规范定义了ORB接口和接口定义语言(IDL)的各种映射规则。
- 可移植。移植ORB到新的操作系统和编译平台应该尽量少地对ORB产生影响。例如, ORB的传输机制必须既能运行在传统的Windows和Linux等平台上, 也能适应运行VxWorks或者LynxOS的传感器和制动器等嵌入式设备上。

怎样才能分解ORB来满足上面这些可修改、稳定、可移植的要求, 并将其按照功能划分为内聚的组呢?

可以使用Layers (108) 来分离ORB中的不同职责, 将其分解为不同组的类, 每一组处理一种类型的功能。

我们把基于CCM的ORB分为4层。顶层提供由OMG定义的标准CCM ORB接口, 代表整个ORB的“应用视图”。第2层提供容器和对象适配器, 它负责管理组件策略, 同时分离和分派客户端请求到组件实例。第3层包括ORB核心, 它实现中间件的连接管理、数据传输、事件分离和并发控制逻辑。底层将ORB的剩余部分屏蔽起来, 使其不用考虑下层操作系统和网络协议的实现细节。

图6-4展示了基于CCM的ORB的分层设计。

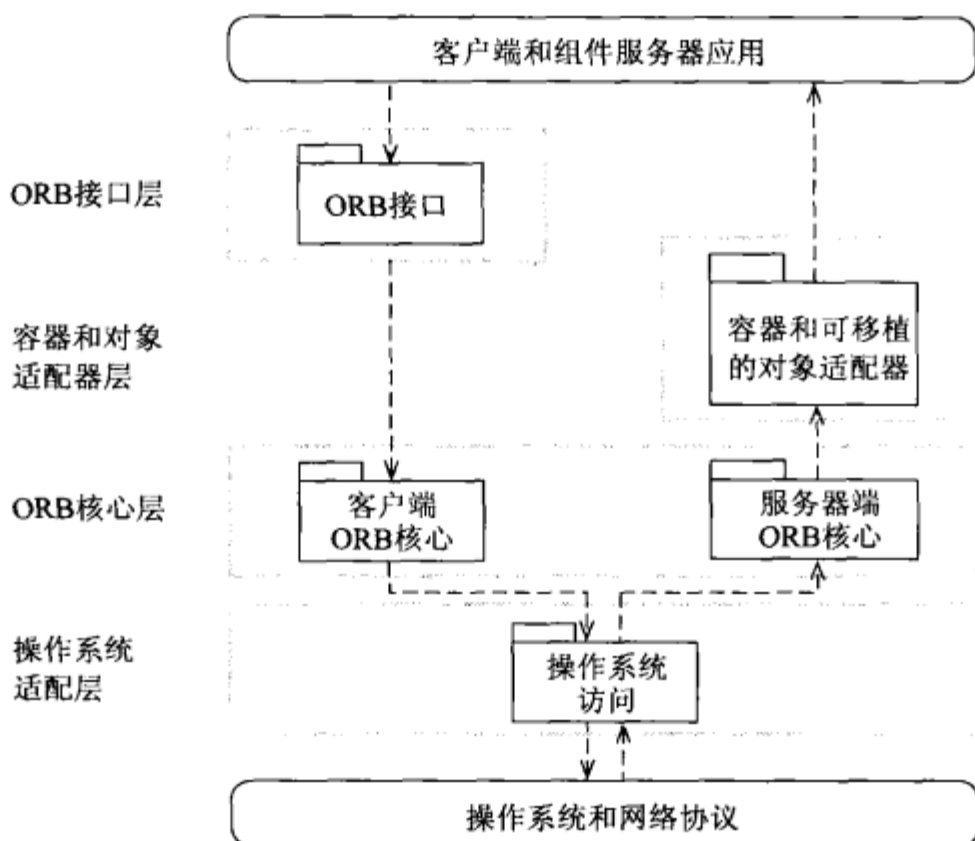


图 6-4

使用Layers模式来实现我们的CCM Broker使得对一层的修改不会影响到其他的层。例如，将对象适配器中的搜索结构从动态散列（dynamic hashing）变为活动分离（active demultiplexing）[PRS+00]不会影响到任何其他的层。Layers也增强了稳定性，因为多个高层的客户端和服务端应用程序有定义良好的对底层网络编程服务的接口，这个接口由ORB提供。而且Layers简化了将ORB移植到新的操作系统的工作，因为移植过程中不会影响应用部分的代码，甚至ORB的实现都不需要做大的改动。

6.3 封装底层系统机制

通信中间件的一个作用是将应用与操作系统和网络特性隔离开，因为后两者都是经常变化的。基于CCM的ORB中间件开发者——与应用开发者不同——负责的是处理底层细节，比如分离事件、跨一个或多个网络接口发送或接收请求以及创建线程并行的执行请求。然而，开发这一层的中间件可能会很难，尤其是当用到使用C写的底层系统API的时候。工作在这一层的ORB开发者经常会遇到以下问题。

- 需要精通数个操作系统平台的细节知识。使用系统级的C语言API要求中间件开发者必须处理不可移植的、冗长的、易错的操作系统特质，比如使用弱类型的套接字句柄来定义通信端点（endpoint）。而且，这些API是不能跨操作系统平台移植的。例如，Windows、Linux和VxWorks分别有不同的线程API，同时对于套接字和事件分离的语义也有微妙的区别。
- 维护的难度增加。构建ORB的方式之一是通过在ORB源代码中显式地应用条件编译指令

(conditional compilation directives) 来处理移植性问题。不过, 在所有用到的地方使用条件编译来指定特定的平台会增加源代码的复杂性。维护和扩展条件编译代码尤其困难, 因为与特定平台相关的细节散落在ORB实现文件的各个角落。

- 不协调的编程范式。系统机制通过C风格的函数调用访问, 会导致与Java、C++、C#等面向对象语言支持的编程风格之间的“阻抗不匹配”。

在实现ORB中间件的时候, 如何避免直接访问底层系统机制呢?

使用Wrapper Facade (269) 来组织ORB的操作系统适配层, 使用简洁而内聚的面向对象类的接口来封装系统编程API和系统机制。

Wrapper Facade提供类型安全的、模块化的和可移植的类接口, 封装底层系统和网络编程机制, 比如套接字、事件分离、同步和线程。总的来说, 在已有的系统级API不可移植或者类型不安全的情况下我们应该使用Wrapper Facade。

ORB实现为了提升其健壮性和可移植性, 它通过Wrapper Facade来访问所有的系统机制, Java Virtual Machine (JVM) [LY99]和ACE C++网络编程工具包[SH02]都提供了这方面的功能。ACE使用类型安全的面向对象接口封装了本机的OS并发、通信、内存管理、事件分离和动态链接机制, 如图6-5所示。

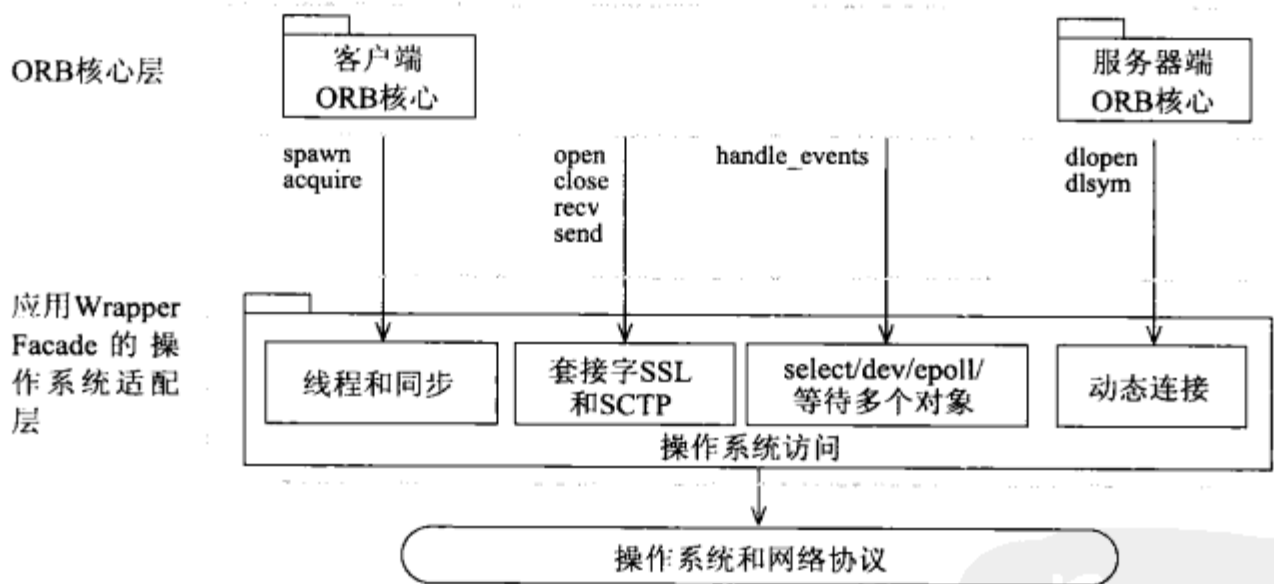


图 6-5

JVM和ACE的Wrapper Facade封装提供了一致的面向对象编程风格, 这使得ORB不需要直接访问弱类型的C系统编程API。标准的编译器和语言处理工具可以在编译时检测类型系统违规, 而不需要等到运行时才能发现错误。因此, 我们可以更容易地维护ORB, 或者将其移植到新的操作系统和编译平台。

6.4 分离 ORB 核心事件

ORB核心的一个职责就是分离来自多个客户端的I/O事件, 并将它们分发给相关的事件处理

者。例如，服务器端ORB核心监听新的客户端连接，读取来自连接客户端的General Inter-ORB Protocol (GIOP) 请求 (request) 信息，并将GIOP应答 (reply) 信息返回给客户端。为了确保对多个客户端的响应能力，ORB核心通过操作系统的select、/dev/epoll、WaitForMultipleObjects和线程等事件分离机制，在多个套接字句柄上等待连接、读写等事件的发生。下面这些问题会给中间件中这一层的开发带来困难。

- 硬编码的事件分离器。开发ORB时所用的一个方法是“硬编码”，其使用单一的事件分离机制，比如select。然而，依赖单一的事件分离机制是有问题的，因为没有哪一个机制是对所有的平台和应用需求都是最高效的。例如，WaitForMultipleObjects在Windows上比select高效，而/dev/epoll在Linux上比select高效。
- 事件分离和事件处理代码紧耦合。开发ORB核心时所用的另一个方法是将事件分离代码和事件处理代码（如GIOP协议处理代码）紧耦合。然而，这使得我们无法在其他使用通信中间件的应用（比如Web server[HPS97]和video-on-demand应用[MSS00]）中像使用黑盒子组件一样使用这段事件分离的代码。而且，如果要引入新的ORB线程策略和请求调度算法，ORB核心的大部分都必须重写。

如何将ORB的实现与单一的事件分离机制解耦合，并且将分离代码和事件处理代码解耦合呢？

使用Reactor (150) 来降低耦合，增加ORB核心的扩展性，通过支持分离和分发多个事件句柄，这些事件句柄由从多个客户端并发到达的事件触发。

Reactor模式通过集成相应的事件处理程序的事件分离和分发，简化了事件驱动的应用程序，尤其是通信中间件。总的来说，如果应用程序或者中间件自己需要处理从多个客户端并发的定时事件，就应该使用Reactor，以防止过紧地耦合到某个底层机制（比如select）上面。

图6-6展示了一种服务器端ORB的实现方式。在这个实现中，我们使用Reactor作为Broker (137) 架构的Invoker (142) 中Server Request Handler (144) 的一部分来驱动ORB核心中的主事件循环。

在这个设计中，组件服务器进程在ORB核心的Reactor实例中初始化一个事件循环，该事件循环阻塞在我们为其配置的事件分离机制上，直到在某个或者某几个通信端点上出现了请求事件。当GIOP请求事件出现的时候，Reactor将请求分配到合适的事件处理程序上，每个事件处理程序都是一个GIOP ConnectionHandler类的实例，每个实例关联到一个连接好的套接字上。然后Reactor调用ConnectionHandler的handle_event方法读入请求并将其传递给容器和对象适配器层。这一层接着把请求分配到合适的组件实例的上行调用方法上，并分派这些上行调用方法。

通过将ORB核心的事件处理部分和底层操作系统的事件分离机制解耦合，Reactor模式可以提高ORB的扩展性。比如，在Windows上我们可以使用WaitForMultipleObjects事件分离系统功能，在UNIX和Linux上则可以使用select或者/dev/epoll机制。同样地，Reactor模式可以使用Java中的SelectableChannels上的Selector实现。Reactor还简化了集成新的事件处理程序的工作。比如，添加一个新的使用PROFibus协议的连接处理器来和我们的仓库管理系统中非CCM部分通信并不会影响Reactor类的接口。

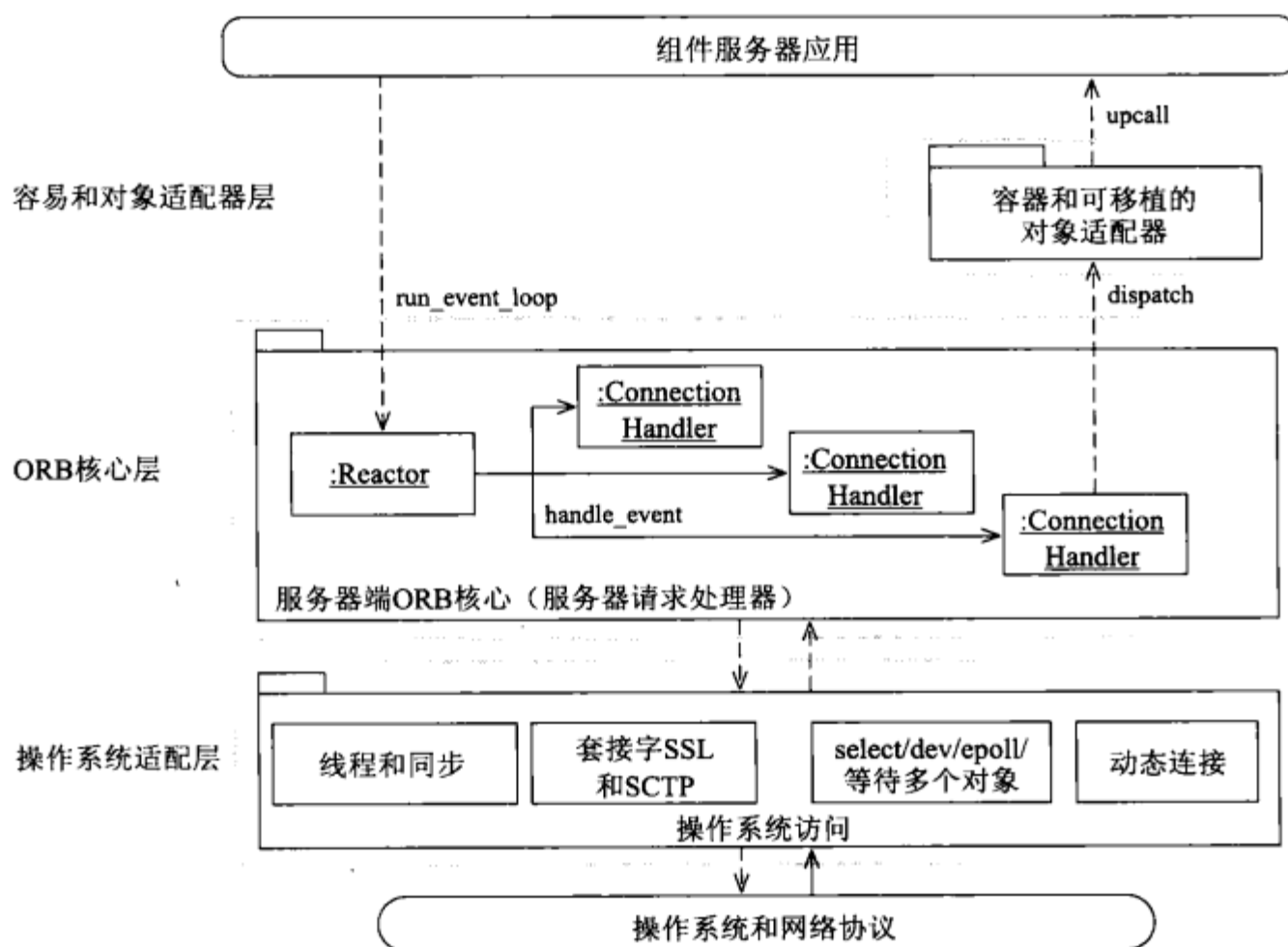


图 6-6

应激性事件分离可能并不是在一个支持高效的异步I/O的操作系统上实现ORB的最具有扩展性的方式。特别地，我们可以在一些平台上使用Proactor (152) 模式实现一个高效的ORB，这个模式构建的事件驱动的并发应用可以从多个客户端异步地接收和处理请求。例如，在Windows上我们可以使用AcceptEx、ReadFile和WriteFile系统功能实现Proactor模式来异步地处理TCP连接和GIOP请求。当这些异步操作完成之后，Windows将结果交给ORB，后者根据这些结果执行相应的操作，然后返回到事件循环中去。

Proactor的主要的优势在于其在实现了高效异步I/O的平台上具有良好的扩展性。其不利条件是现在能正确而高效地提供这种支持的平台还相对较少。我们的目标是设计一个可移植的ORB，所以我们选择了Reactor模式作为基本的事件分离机制。

6.5 ORB 连接管理

连接管理是ORB核心的另一个关键职责。例如，实现了GIOP的ORB核心必须负责建立TCP连接，并为每个TCP服务器端点初始化协议处理器。我们将连接管理逻辑限制在ORB核心内，这样应用组件就可以专注于处理应用相关的请求和回复，而不必关注底层的操作系统和网络编程方面的任务。

然而，ORB核心并不局限于GIOP和TCP[OKS+00]两种传输方式。虽然TCP可以可靠地传输GIOP请求，但是它在流控制和拥塞控制算法方面的缺陷却限制了它在仓库管理的传感器和制动

器等对时间要求严格的组件上的应用，这时候流控制传输协议（SCTP）或者实时协议（Real-Time Protocol, RTP）或许更为合适。如果客户端和组件实例位于同一个终端系统上，而且这个系统的操作系统支持共享内存的话，则共享内存传输机制的效率往往要比别的方式要高。为了保证数据的完整性和机密性，我们可能需要通过加密的安全套接字层（SSL）连接来交换请求和响应。所以，ORB核心必须具有这种灵活性，支持多种传输机制。

CCM引用架构将由ORB核心执行的连接管理任务和由应用组件执行的请求处理进行了显式的解耦合。然而，实现ORB内部连接管理的常见方式是使用套接字这样的底层网络API。相似地，ORB连接建立协议和其通信协议也经常紧密地耦合在一起。

不幸的是，使用套接字网络编程API来实现ORB连接管理和在TCP/IP连接建立协议中使用GIOP消息格式都使用了硬编码，这会引入两个问题。

- 僵化。如果ORB的连接管理数据结构和算法紧紧地纠缠在一起，要修改ORB核心就会变得非常困难。这样的话，将一个紧耦合的ORB核心移植到新的网络协议和编程API——比如SSL、SCTP、RTP、共享内存或者Windows命名管道——就会花费很多时间。例如，如果一个ORB核心已经和底层的套接字API紧紧地耦合在一起了，要把底层传输机制换成共享内存或者SSL将是非常困难的。
- 低效。如果我们允许ORB和应用的开发人员在产品开发的后期——比如经过广泛的运行时性能分析之后——选择合适的ORB内部策略，就可以做到更好的系统优化。例如，多线程的实时客户端可能需要使用Thread-Specific Storage (228) 来存储传输端点。而CCM组件服务器可能要求每个连接运行在自己的线程中，以便消除为每个请求设置一个锁的消耗。如果连接管理机制和其他的内部ORB策略是通过硬编码的方式紧绑定的，要想使用其他的更高效的机制就会非常麻烦，甚至很多地方需要重做。

那么怎样才能使得ORB核心的连接管理机制能够支持多种传输方式，并且可以在开发周期内任何时候灵活地（重新）配置与连接相关的行为呢？

使用Acceptor-Connector (154) 设计来提高ORB核心连接管理和初始化的灵活性，因为一旦连接的建立和服务的初始化完成之后，这个模式就可以将这些活动与在其后执行的任务解耦合。

在ORB核心的服务器端，Acceptor-Connector模式中的acceptor组件负责被动地建立连接和初始化服务。相反，在ORB核心的客户端，该模式中的connector组件则负责主动地建立连接和初始化服务。我们把Acceptor-Connector模式和Reactor (150) 模式联合起来使用，为我们的ORB创建了一个可插拔的协议框架[OKS+00]。这个框架可以为ORB中支持的各种网络协议执行建立连接和初始化连接处理器的工作，如下所示。

- 客户端ORB核心。为了响应操作调用或者显式地绑定到远程组件实例上，客户端ORB核心使用Connector指向目标服务器ORB并应用相应的协议类型来初始化连接，在初始化连接完成之后再初始化合适类型的ConnectionHandler来服务该连接。
- 服务器端ORB核心。服务器端ORB核心收到来自客户端的连接，然后使用Acceptor创建相

应类型的ConnectionHandler来服务这个新的客户端连接。

如图6-7所示，一旦事件可以处理了，Acceptor和Connector都是事件处理程序，在事件可以开始处理之后它们可以由ORB的Reactor自动分派。

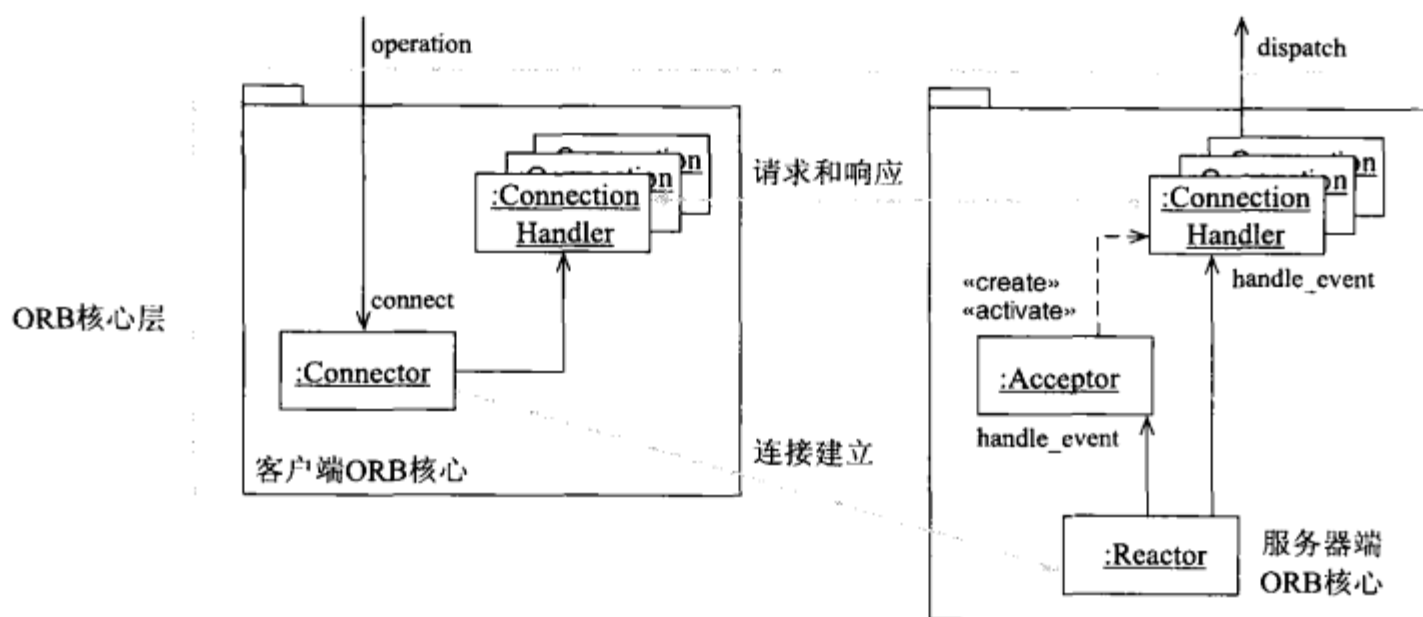


图 6-7

该图显示，如果客户端要调用某个远程的操作，它首先通过其Connector发起一个connect调用，以获得一个连接，并初始化与期望的网络协议相对应的ConnectionHandler。在服务器端ORB核心中，Reactor通过handle_event方法通知Acceptor接收新的客户端连接，并创建相应的ConnectionHandler。在ORB核心中的ConnectionHandler激活之后，它会在这个连接上执行必要的协议处理，并最终将请求通过ORB的容器和对象适配器分派到合适的组件实例。

这个基于CCM的ORB通过联合使用Acceptor-Connector和Reactor，提高了ORB的灵活性，解除了事件分离与连接管理和协议处理的耦合关系。这个设计使得开发人员可以方便地集成最适合于我们的仓库管理流程管理系统的网络协议和网络编程API。

6.6 提高 ORB 的可伸缩性

随着客户端数量的增长，可伸缩的端对端的性能对于处理流量的增加就变得非常重要了。默认情况下，GIOP运行在TCP之上，TCP使用流控制确保发送者不会过快地生成数据，以至于接收者或者拥塞的网络来不及缓存和处理[Ste93]。如果CCM发送者通过TCP发送大量的数据，以至于接收者来不及处理，则连接会自动地启动流控制并阻塞发送者，直到发送者和接收者的速度能匹配得上为止。

在6.4节分离ORB核心事件中，我们给出的基于Reactor模式的设计是在一个单独的控制线程中处理所有的请求。虽然这个设计的实现很直观，但是它会带来以下问题。

- 不可伸缩。由于单线程的应激性ORB服务器一次只能处理一个客户端请求，使用它来处理长周期客户端请求时，其伸缩性是很差的。
- 饥饿。在连接的流控制决定何时向客户端发送回复之前，整个ORB服务器将一直阻塞在

那里,这会导致其他的客户端的请求无法得到处理。

相反,在所有的ORB处理中都使用的多线程对于短周期处理也同样是低效的,因为线程会带来严重的同步、上下文切换和数据移动[PSC+01]等并发控制开销。

ORB如何才能高效地管理并发处理呢?怎样才能让多个长周期请求可以在单个或多个CPU上同时执行,而又能避免处理短周期请求时引入不必要的并发控制开销呢?

使用Half-Sync/Half-Async (209) 并发模型来区分ORB中的短周期处理和长周期处理,这样就既提高了伸缩性,又不会带来过度的并发控制开销。

在基于CCM的ORB中,Half-Sync/Half-Async并发模型使用RequestHandler池在单独的控制线程中并发地处理长周期客户端请求,并作出响应。相反,对于短周期Acceptor连接建立和Request事件处理,则借用Reactor的控制线程在ConnectionHandler中采用应激性的方式处理。

图6-8展示了我们基于Half-Sync/Half-Async的ORB的设计。从这里你可以看到,Reactor是如何通过分派Acceptor的handle_event方法来驱动Acceptor建立连接的。Request事件处理部分地由Reactor驱动,Reactor负责分派ConnectionHandler的handle_event方法来将请求信息读取到缓存中。然后这个缓存被放置到同步的RequestQueue里面,这个RequestQueue用于将请求传递给RequestHandler池,池中的每个RequestHandler可以在自己的控制线程中并发地处理这些请求。

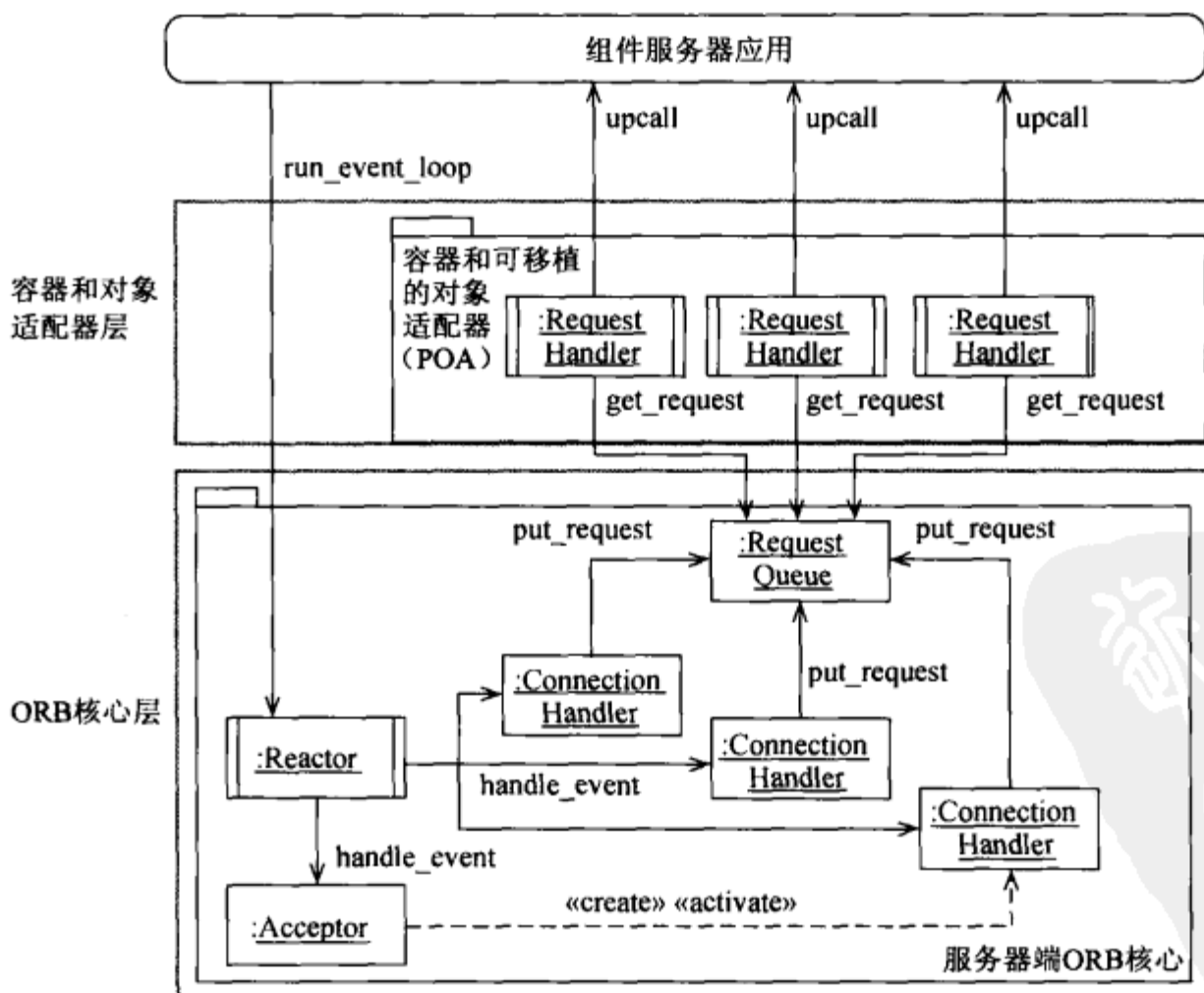


图 6-8

由于我们在ORB中使用了Half-Sync/Half-Async, 所以多个客户端的请求/响应可以在各自的线程中并发地运行, 相对于纯粹基于Reactor的设计, ORB的伸缩性得到了增强。同样, 由于每个线程都能够独立地阻塞, 在向客户端发送回复的时候, 就不需要整个服务器ORB进程等待流控制来清空连接 (clear on a connection)。由于我们的仓库管理流程控制系统中有些子系统更适合基于Reactor的设计, 因此我们的ORB对这两种方式都支持。

Half-Sync/Half-Async模型对于ORB来说并不总是最高效的, 因为在Reactor线程和RequestHandler之间传递请求会引入动态内存分配、大量的同步操作、上下文切换和缓存更新。这可能会给ORB引入大量不必要延迟, 对于短周期请求尤为如此。一种替换的办法是使用Leader/Followers (211) 模式, Leader/Followers模式提供更为高效和可靠的并发模型, 在这种模型中多个线程轮换共享事件源, 比如使用被动方式套接字句柄来检测、分离、分派和处理事件源上出现的服务请求。

Leader/Followers的好处是它可以避免引入一个单独的Reactor线程和同步的RequestQueue [PSC+01], 从而降低系统开销。其缺点是相对于Half-Sync/Half-Async, Leader/Followers的伸缩性要低一些, 因为前者的请求队列是在ORB的虚拟内存上实现的, 而后者则是在操作系统的内核上实现的。因为我们的目标是设计一个高度扩展性的ORB, 我们选择了Half-Sync/Half-Async作为基本的并发机制。

6.7 实现同步请求队列

Half-Sync/Half-Async (209) 的核心是一个RequestQueue排队层。在我们的基于CCM的ORB中, 异步 (应激) 层中的ConnectionHandler是“生产者”, 它将客户端请求插入到RequestQueue中。而同步 (多线程) 层中的RequestHandler池是“消费者”, 它把客户端请求移出队列并处理。

如果RequestQueue的实现过于简陋会造成以下问题。

- 由于在Half-Sync/Half-Async模式中不同层中包含多个生产者和消费者线程, 如果它们对RequestQueue的并发访问不能够正确地序列化而出现了竞争条件, 就会破坏RequestQueue的内部状态。
- 如果使用的互斥锁过于简单, 当队列为空或者满的时候就会导致生产者和消费者线程进入“忙等待”状态, 无谓地消耗CPU周期。

当不同层中的线程同时向RequestQueue插入和移除客户端请求的时候, 怎样才能避免出现竞争条件或者忙等待呢?

将RequestQueue实现为Monitor Object (214) 来序列化并发的方法调用, 从而保证一次只有一个方法运行, 这样put_request和get_request的方法就可以相互配合调度其执行顺序, 避免生产者和消费者线程在RequestQueue满或者空的时候出现忙等待的状况。

同步的RequestQueue使用Monitor锁来序列化对Monitor对象的访问, 并使用POSIX Pthreads或者java.util.concurrent.locks的条件变量来实现队列的非空和非满Monitor状态。这个同步的RequestQueue可以集成到ORB的Half-Sync/Half-Async实现中, 如图6-9所示。

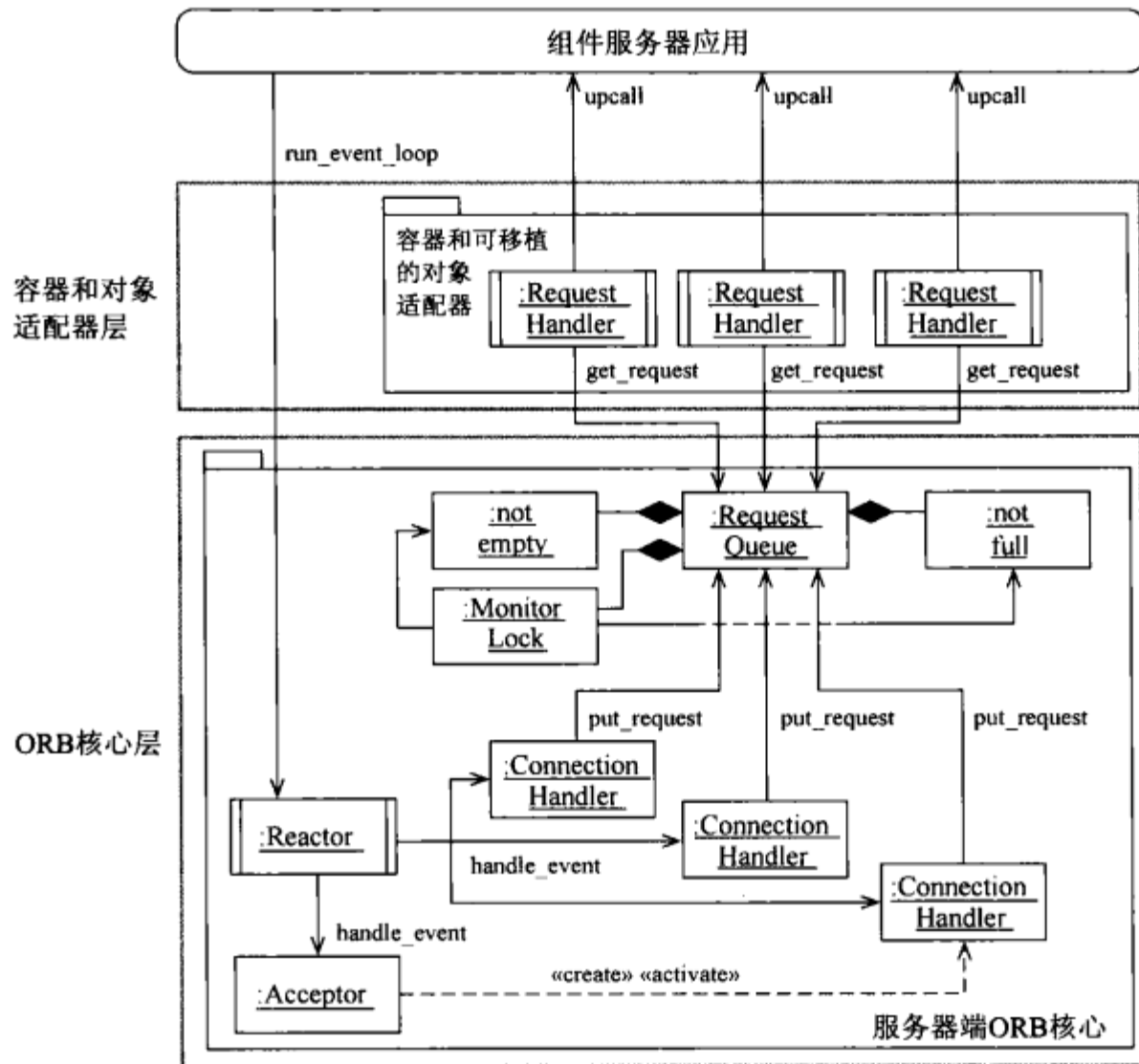


图 6-9

当运行于RequestHandlers池中的某个消费者线程试图从一个空的RequestQueue里获取客户端请求时，队列的get_request方法自动释放Monitor锁，线程自动将自己挂起到非空的Monitor条件上。直到某个运行于生产者线程的ConnectionHandler将一个客户端请求放入队列，且RequestQueue不再为空时，消费者线程才继续执行。

Monitor Object通过为在协作的线程（在这些线程中，对象同步相当于方法调用）之间共享RequestQueue而提供一个简洁的编程模型，简化了我们的ORB的Half-Sync/Half-Async并发设计。同步的put_request和get_request方法根据RequestQueue的Monitor条件来决定是应当挂起还是继续执行。

6.8 可互换的内部 ORB 机制

通信中间件往往需要支持各种领域的应用需求，相应地，其运营环境也千差万别。为了满足这些不同的需求和环境，ORB需要支持多种内部机制实现。可替换的并发模型、事件和请求分离器、连接管理和数据传输，以及（解）封送架构都是这方面的例子。

要支持ORB内部机制的多种实现,一种方式是在编译时使用预处理宏和条件编译来静态地配置ORB。比如,as/dev/epoll和WaitForMultipleObjects功能只在特定的操作系统上可以使用,那么你的ORB的C或者C++源代码可能得使用`#if ... #elif ... #else .. #endif`等条件编译块。预处理器在编译的时候就会检查这些宏的值,并根据这些值选择合适的事件分离机制。

虽然有很多的C/C++ ORB都使用了这种方式,但是它确实存在一些问题。

- 僵化。预处理宏所能配置的机制仅限于编译时已知的机制,这使得要想配置ORB来支持基于启动时或者运行时知识而选择的机制变得非常困难。例如,ORB可能希望动态地根据当时的CPU的数目、负载或者某个网络协议的可用性来进行自我配置,以便使用不同的并发模型或者传输机制。
- 易出错。使用预处理宏和条件编译使得我们很难理解和验证ORB。尤其是,对于ORB行为和状态的修改会分散在源代码的各个角落,使得ORB难于编译,更难于测试到代码的所有路径[MPY+04]。

那么,怎样才能使ORB灵活地替换器内部机制,封装每个机制的状态和行为,以便对于其中的一个修改不至于扩散到整个ORB呢?

使用Strategy (266) 配置来支持多个可透明“插拔”的ORB机制,通过提取各种机制中的共同点,并将策略的名字与其行为和状态显式地关联起来。

我们的基于CCM的ORB很多地方都使用了Strategy,用以抽取内部机制的因子,这些机制在传统的ORB中往往是硬编码的。图6-10展示了我们的ORB提供的策略钩子,它们使用运行时多态来简化不同的(解)封送、请求和事件分离、连接管理、客户/服务器数据传输和并发机制的动态(重新)配置。

编译时多态也可以使用特化(specialization)技术来实现,比如部分求值、切面织入等,这些技术虽然不如运行时多态灵活,但是其效率却更加高[KGS+05]。

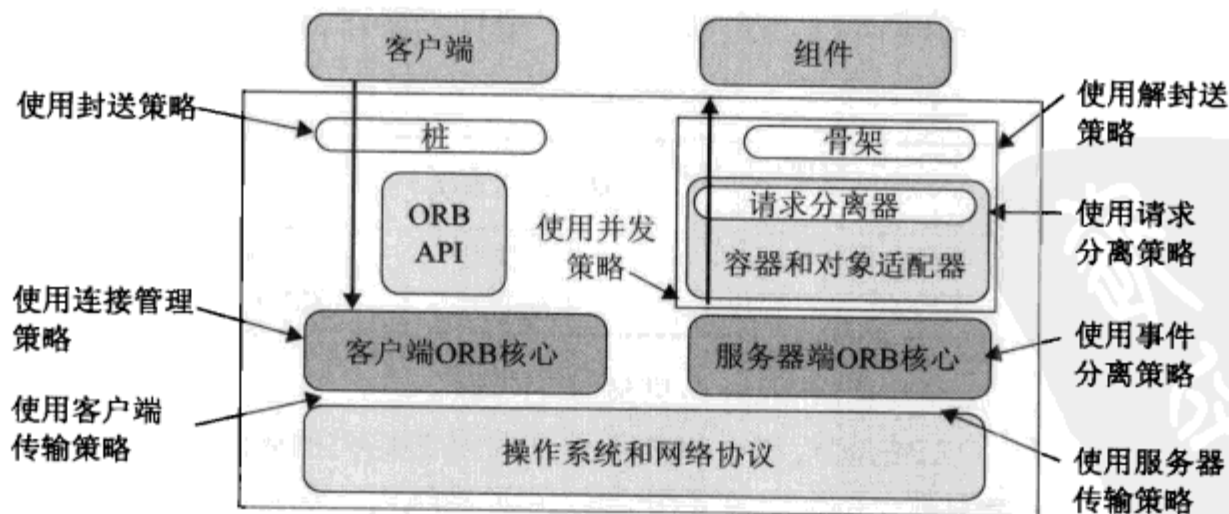


图 6-10

在我们的基于CCM的ORB里面应用Strategy移除了对于ORB内部机制实现的直接依赖(lexical dependencies),不论最终配置的是哪种机制我们总是通过基类的接口来访问的。而且,

Strategy简化了对ORB行为的自定义，它使得我们可以在启动的时候——甚至推迟到运行时——进行动态地配置，而不是只能在编译时完成静态配置。

Template Method (265) 模式和Strategy一样也可以支持多种可透明地“插拔”的ORB机制。这里，我们选择了Strategy，因为它使用了委派的方式，这样我们就可以动态地（比如在初始化期间）选择和/或替换ORB机制。而Template Method使用继承的方式，因此对于ORB机制的选择将绑定在编译时，这对于我们的应用来说限制过于严格。

6.9 管理 ORB 策略

我们的基于CCM的ORB在各个层次上支持多种策略。

- 桩和骨架支持各种(解)封送策略，比如Common Data Representation (CDR)、eXternal Data Representation (XDR) 和其他适合于ORB的私有策略，它们可以跨同质的硬件、操作系统和编译器通信。
- 容器和对象适配器层支持多种请求分离策略，比如动态散列、完美散列和主动分离(active demultiplexing) [GS97a]，同时也支持多种生命周期策略，比如会话容器或者实体(entity)容器。
- ORB核心层支持多种事件分离策略，比如用select、/dev/ep11、WaitForMultipleObjects实现的Reactor，或者专属于某个VME（虚拟机环境）的分离器；同时也支持多种连接管理策略，比如进程范围(process-wide)缓存的连接和专属于某个线程的缓存连接；还支持不同的ConnectionHandler并发策略，比如单线程的应激性的或者多线程的Half-Sync/Half-Async；还支持不同的传输策略，比如TCP/IP、SSL、SCTP、VME和共享内存。

表6-1展示的两组策略分别用于创建仓库管理流程控制ORB中两个不同的子系统的配置。

- 第一个用于部署在VxWorks的嵌入式设备上的传感器和制动器。
- 第二个用于部署在Solaris或者Linux的服务器上的仓库业务逻辑和基础设施功能。

表 6-1

应用	并发策略	封送和解封送策略	请求分离策略	协议	事件分离策略
传感器和制动器	应激性	专用的	完美散列	VME后机	VME专用分离器
仓库（业务逻辑）	Half-Sync/Half-Async	CDR	主动分离	TCP/IP	基于select的分离器

然而，在ORB中如此彻底地使用Strategy会导致以下问题。

- 复杂的维护和配置。ORB的源代码中可能会遍布着对策略类的硬编码引用，这使得维护和配置都会变得异常困难。例如，在某些特定的子系统内，如传感器和制动器或者业务逻辑，众多独立的策略必须能够协调工作。单独通过名字确定这些策略，将一个领域的策略替换为另一个领域内潜在的不同的一套策略，会引入大量繁冗的工作。
- 语义不兼容。并非所有的策略都可以语义兼容地交互。例如，专属于某个虚拟机环境的

事件分离策略可能无法和TCP/IP协议协同工作。而有些策略则仅在某些特定条件满足的时候才能使用。例如，完美散列分离只能用于所有的组件实例都可以离线静态注册的系统[GS97b]。

那么，一个高度可配置的ORB要将多套策略组合在一起，怎样才能达到既降低管理众多策略的复杂性，又能保证语义上的兼容性呢？

引入Abstract Factory (311) 将多个ORB策略组织成数个可管理的语义上兼容的配置。

前面我们所有的专属于某个客户端或者服务器的ORB策略都封装进不同的抽象工厂。通过使用Abstract Factory模式，提供一个集成了所有用于配置ORB的策略的唯一访问点，应用开发人员和最终用户可以配置内部机制，构成不同类型的、语义兼容的ORB。具体子类^①则将专属于某个应用或者领域的语义上兼容的策略组织在一起，使它们可以相互协作。

图6-11展示了某抽象工厂的两个实例，这两个实例分别用于为业务逻辑和传感器/制动器子系统中运行的应用配置ORB。

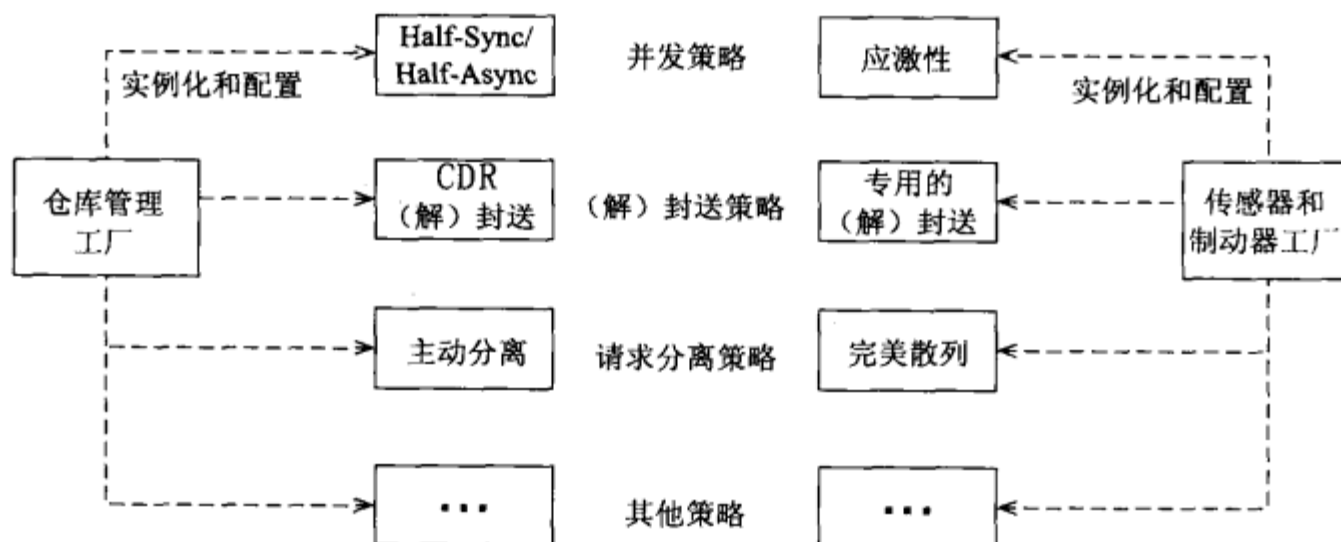


图 6-11

Abstract Factory帮助我们简化了ORB的维护和配置工作。针对不同的仓库管理流程控制子系统，语义上兼容的ORB策略被组织在一起，并确保它们一起切换。

6.10 ORB 动态配置

虽然内存、CPU等计算资源越来越便宜。然而，这些系统资源毕竟是有穷的，尤其是对于实时系统和嵌入式系统而言，往往要求内存消耗要少，CPU处理开销要尽量地低[GS98]。与之类似，很多应用也可以通过支持运行时配置策略，而从ORB的动态扩展能力获益。

虽然Strategy (266) 和Abstract Factory (311) 让我们可以比较方便地为某个特定的应用需求和系统特征对ORB进行自定义，这些模式仍然会导致下列问题。

^① 这里是指抽象工厂的子类。——译者注

- 过度的资源消耗。过度地使用Strategy模式，你会发现ORB中要配置的内部机制越来越多，这就需要更多的系统资源来运行ORB及其应用。
- 不可避免的系统停机。如果策略是在构建时使用Abstract Factory静态配置的，要想改进已有的策略或者更换新的策略，你往往需要为“消费者”修改已有的策略或者Abstract Factory的源代码，重新编译和重新连接ORB，重启正在运行的ORB及其应用的组件的实例，这样才能够成功地部署新的功能。

总的来说，静态配置只适用于较少的固定数目策略的情况。在更为复杂的、可扩展的ORB上使用这个技术，会增加维护的复杂性，提高系统资源消耗，要求系统停机来增加或者更换已有的组件实例。

对于普遍应用的Strategy和Abstract Factory模式，如何能够在ORB实现中合理地使用，而降低“过大、静态”的副作用呢？

引入Component Configurator (289) 来动态地将自定义的策略和Abstract Factory对象在启动时或者运行时连接进ORB或者从ORB解除连接。

我们的基于CCM的ORB使用了Component Configurator来实现运行时配置Abstract Factory，这些工厂中包含了多组语义上兼容的策略。ORB的初始化代码根据特定的用例使用由操作系统平台提供和/或由ORB的操作系统适配层的Wrapper Facade封装的动态配置机制来连接到合适的工厂。常用的动态配置机制包括UNIX中的dlopen/dlsym/dlclose系统函数，Windows中的LoadLibrary/GetProcAddress系统函数，以及Java中的Applet设施（facility）。通过联合使用Component Configurator和这些系统函数，我们可以将ORB的行为与何时对其内部实现机制进行配置解除耦合，而且同时保证这些配置策略的语义兼容性。

ORB策略从动态链接库（DDL）链接进ORB，可以在编译期完成，也可以在启动的时候，甚至是运行时。而且，Component Configurator还可以减少ORB的内存使用，因为应用开发人员可以只在某些特定的情况下，才把必要的策略动态地连接到ORB中。图6-12展示了两个工厂，一个用于我们的仓库管理流程控制系统中的业务逻辑子系统，另一个用于传感器和制动器子系统。

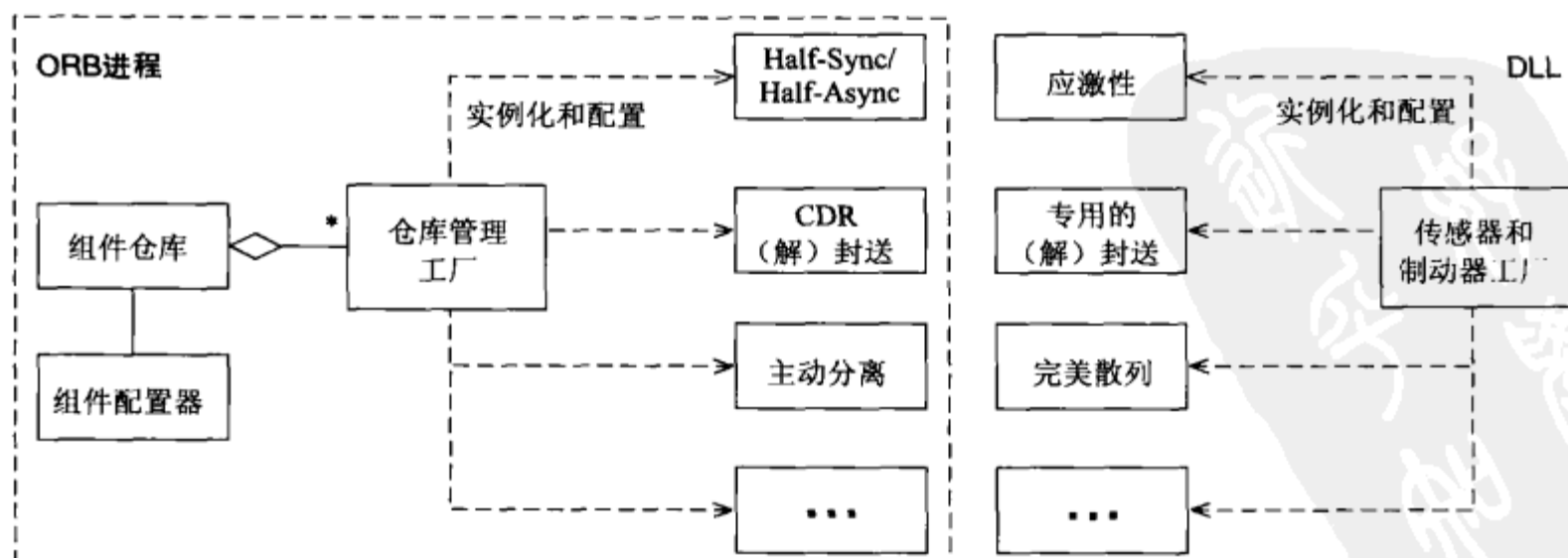


图 6-12

在这个配置中，WarehouseBusinessLogicFactory是装载在当前ORB进程中的。使用ORB配置的应用将使用选定的ORB并发策略、（解）封送策略、请求分离策略等来完成应用的处理过程。相反，SensorAndActuatorFactory则保存在当前ORB进程之外的一个DLL中。通过使用Component Configurator，这个工厂可以在ORB进程开始运行时动态地装载进来。

在ORB进程中，SensorAndActuatorFactory是由ComponentRepository维护的，ComponentRepository负责管理当前ORB中装载的可配置的组件实例。ComponentConfigurator使用ComponentRepository来协调组件实例的（重新）配置，比如，它可以将当前的SensorAndActuatorFactory解除链接，并链接一个经过优化的版本。

Component Configurator允许应用开发人员动态地配置ORB的行为，或者通过裁剪ORB来满足特定的运营环境和应用需求。除了增强了灵活性，这个模式还可以避免由那些没有用到的策略所引入的时间和空间上的消耗。而且，Component Configurator还可以让应用开发人员不必访问或者修改ORB的源代码就可以配置ORB，而且经常可以对行为的某些方面进行升级时也不必关掉整个ORB系统。

6.11 通信中间件总结

本章描述的基于CCM的ORB是模式序列的一个产品。我们的模式序列使用一种良好定义的、经过时间验证的风格为读者呈现了基本的ORB机制，比如同步、传输、请求和事件分离，以及封送和解封送。我们最关键的设计目标是保证ORB可配置、可扩展、可适应和可移植。为了实现这样的设计目标，我们的模式序列中的模式需要作出精心的选择、集成和实现，这些都是基于以往对这些模式的使用经验做出的。它们包括其他的标准中间件，比如Web服务器[POSA2]^①[HMS97]；面向对象网络编程框架，比如ACE[SH03]；网络应用，比如应用层网关（application-level gateway）[Sch00]，电子医疗成像系统（electronic medical imaging system）[PHS96]和航电使命计算系统（avionics mission computing systems）[SGS01]。该序列中前两个模式——Broker (137) 和Layers (108)——定义了基于CCM的ORB的核心结构。Broker将应用功能从通信中间件功能中分离出来，而Layers则按照各自的抽象层次对不同的通信中间件服务进行分层。

序列中第3个模式是Wrapper Facade (269)，它帮助我们构建ORB的最底层设计，即操作系统抽象层。它把操作系统抽象层构建为模块化的可独立使用的构建块。每个Wrapper Facade为操作系统的特定职责或一组功能提供了一个有意义的抽象，它将相应的API函数封装成类型安全的、模块化的、可移植的类。实现ORB高层的时候就不需要显式地依赖于某个特定的平台。

序列中接下来的一组模式是关于服务器端ORB核心层的。从Broker架构的角度来看，服务器端ORB核心扮演的是Invoker (142) 的角色，它使用Server Request Handler (144) 来从网络上接收消息和请求，并把这些消息和请求分派给相应的组件实例来做进一步的处理。Reactor (150) 为Server Request Handler提供了分离和分派的基础设施，以便它通过可扩展的方式应对不同的事件处理策略，而这种分离机制独立于底层分离机制——比如select和WaitForMultipleObjects。Acceptor-

^① [POSA2]第1章描述了实现一个Web服务器的模式序列，那里包含了本章所描述的多个模式。

Connector (154) 通过引入特化 (specialized) 的事件处理程序帮助Reactor初始化和接收网络连接事件, 从而将连接建立从ORB核心内的通信中分离出来。Reactor结合使用了Half-Sync/Half-Async (209) 和Monitor Object (214), 以便可以并发地处理客户端的请求, 以此提高服务器端ORB的可伸缩性。

序列中最后3个模式是关于可配置性的。我们在ORB的机制可能发生变化的地方都使用了Strategy (266), 比如连接管理、并发和事件/请求分离机制。为了给ORB配置一组语义上兼容的策略, 客户端和服务端端的实现都使用了Abstract Factory (311)。这两个模式的联合使用, 使得我们可以方便的创建ORB的变体, 以满足特定用户和应用场景的需要。Component Configurator (289) 专门用于更新ORB内的策略和抽象工厂, 而不需要修改已有的代码、重编译或静态重链接已有的代码, 或者停止并重启已有的ORB和应用组件实例。

表6-2总结了特定ORB设计挑战和我们用于应对这些挑战的模式序列的对应关系。

表 6-2

模 式	挑 战
Broker	定义ORB的基线架构
Layers	结构化ORB内部设计以支持重用, 并对关注点进行分离
Wrapper Facade	封装底层系统功能, 提高可移植性
Reactor	高效地分离ORB核心事件
Acceptor-Connector	高效地管理ORB连接
Half-Sync/Half-Async	通过并发地处理请求, 提高ORB的可伸缩性
Monitor Object	高效地同步Half-Sync/Half-Async请求队列
Strategy	透明地替换内部ORB机制
Abstract Factory	将ORB机制策略组织成语义兼容的组
Component Configurator	动态地配置ORB策略

对这个模式序列的分析告诉我们, 它不仅可以帮助我们做出满足仓库管理流程控制系统需求的设计, 而且这个设计是可配置的, 所以它也可以满足众多其他领域分布式系统的需求。尤其是, 我们不但针对这个特定的技术问题——通信中间件——创建了一个产品线架构, 而且这个架构是位于一个更大的为仓库管理流程控制这个应用领域所创建的产品线架构之中的。因此, 基于我们的模式序列所做出的ORB的架构和实现是可扩展的, 也是可重用的, 它不是仅针对仓库管理领域的应景之作, 在其他领域也可以高效地应用。

所以, 就不奇怪本章所描述的序列构成了几个ORB的基础, 包括Component-Integrated ACE ORB (CIAO) [WSG+03]、The ACE ORB (TAO) [SNG+02]和ZEN [KSK04]。CIAO和TAO创建了一个基于标准的C++通信中间件平台。它组合了轻量级CCM[OMG04b]特性——比如标准的特化 (specifying)、实现、打包、装配 (assembling) 和组件实例部署的机制, 以及实时CORBA (Real-Time CORBA) [OMG03a][OMG05a]特性, 比如线程池、可移植的优先级、同步、优先级预留策略和显式绑定机制。ZEN也是一个实时CORBA的实现, 它使用了实时Java (Real-Time Java) [BGB+00]特性, 比如作用域内存 (scoped memory) 和实时线程。

CIAO、TAO和ZEN是开源的[DOC]，并且已经应用于很多商用分布式系统，包括航空电子(avionics)和vehtronics、工厂自动化和流程控制、电信呼叫处理、交换、网络管理和医疗工程和成像。这些系统大多都有实时性的要求以满足他们严格的计算时间、执行周期、带宽/延时方面的需求。它们的设计非常灵活，并且是基于模式的，所以它们也适用于要求传统的、“尽力而为”的支持的分布式应用。应用那些关注于性能和可配置性的模式，以及本章所描述的模式序列，有助于我们为CIAO、TAO和ZEN创建一个产品线架构，该架构既能满足这些需求，同时不失其简洁性和易懂性。

关于TAO和ZEN中有关这些模式的进一步信息，请参考[SC99][CSKO+02]和[KSS05]。

拓扑是一种辅助手段，探知所有经验系统的价值；
拓扑是一门科学，展现事物的基本模型和基本关系。

——Buckminster Fuller

我们模式故事的第2章是关于仓库管理流程控制系统中的一个特定领域对象——仓库拓扑的。本章我们描述了仓库拓扑的内部设计，包括仓库物理存放设施的表现和访问，为适应仓库特定需求对存放结构进行的扩展和调整，以及仓库拓扑设计与系统基线架构的集成。

7.1 仓库拓扑基线

对仓库管理流程控制系统来说，仓库拓扑Domain Object (121) 的主要职责是提供物理仓库结构的表现形式给其他领域对象，主要包括仓库管理和物流控制对象。系统的基线架构定义了仓库拓扑领域对象的基础结构，可划分为Explicit Interface (163)、Encapsulated Implementation (181)、Half-Object plus Protocol (188) 分布式架构，以及Active Object (212) 并发模型。

7.2 表现层次化的存储结构

要弄清楚仓库拓扑领域对象的封装实现，首先要面对的问题是如何对物理存储的层次结构进行建模、传送设施以及二者间的联系，以恰当地表现出现实世界中的任意仓库结构，比如不同的仓库大小、存储组织以及可用的存储单元类型。随着时间的流逝，仓库的组织结构还可能发生变化，例如对其进行扩展或现代化改造，或者是安装了原来没有的新的存储类型。仓库结构对应的软件模型必须支持这种重新组织而不会影响没有变化的任何部分。

我们怎样才能能在仓库拓扑Domain Object (121) 中表现现实世界中的任意存储结构，并支持灵活的重组和改进呢？

采用Composite (185) 方式来对仓库结构的层次化组织进行建模，以便于客户端的透明使用。

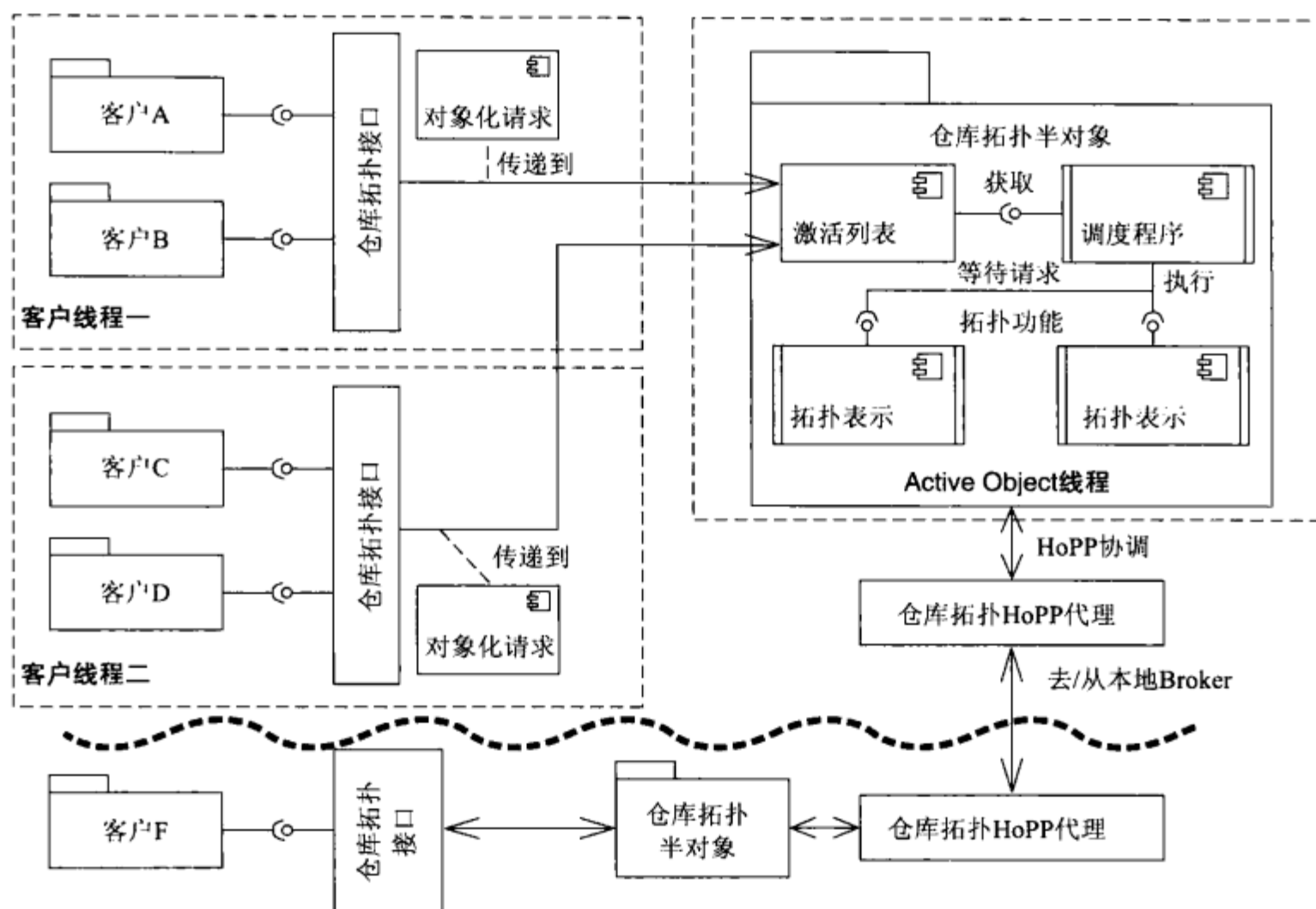


图 7-1

定义一个Storage类表示拓扑结构的根类，并提供Explicit Interface (163) 为某一具体存储层次结构中的所有元素所共享，如存储和提取物品、检查可用库存容量，以及访问已存物品信息等。从存储类（Storage）派生了两个类：AggregateStorage提供对复合存储类型通用的数据结构和方

法，如通道；AtomicStorage提供对基本存储类型通用的数据结构和方

法，如箱子。所有具体的存储类型都从这两个类继承得到，如代表聚合存储类型的Aisle、Side和Rack类，以及代表原子存储类型的Bin、TransferBin、Door，以及Dump类等。

传输设施也无缝地集成至此结构中。就其本质而言，传输设施只是一个特殊的存储类，可以用来存储物品、提取物品、检查其可用容量、访问其存储物品的信息等。传输设施和“真正的”存储单元唯一差别在于它是移动和主动的，而不是固定和被动的，叉车用来提取和存储物品，而箱子则用来存放物品和从中取出物品。另外，也有聚合的传输设施，比如有几节车厢的火车。

传输设施和物理存储单元可以通过两个Storage实例之间的transportationFacility关系彼此联系起来，如果某一特定传送设施为特定的一个（一套）物理存储单元服务，比如吊车为某一通道中所有箱子和传送箱服务，这可以通过将相应的吊车和通道对象在层次结构中联系起来表示。

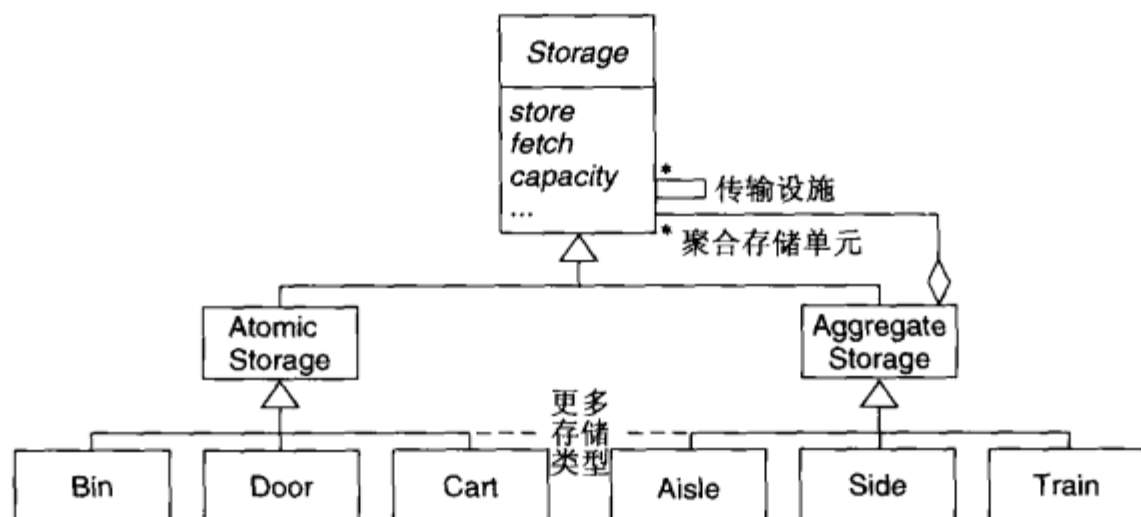


图 7-2

Composite直接支持创建任意的存储层次结构，如图7-3所示。

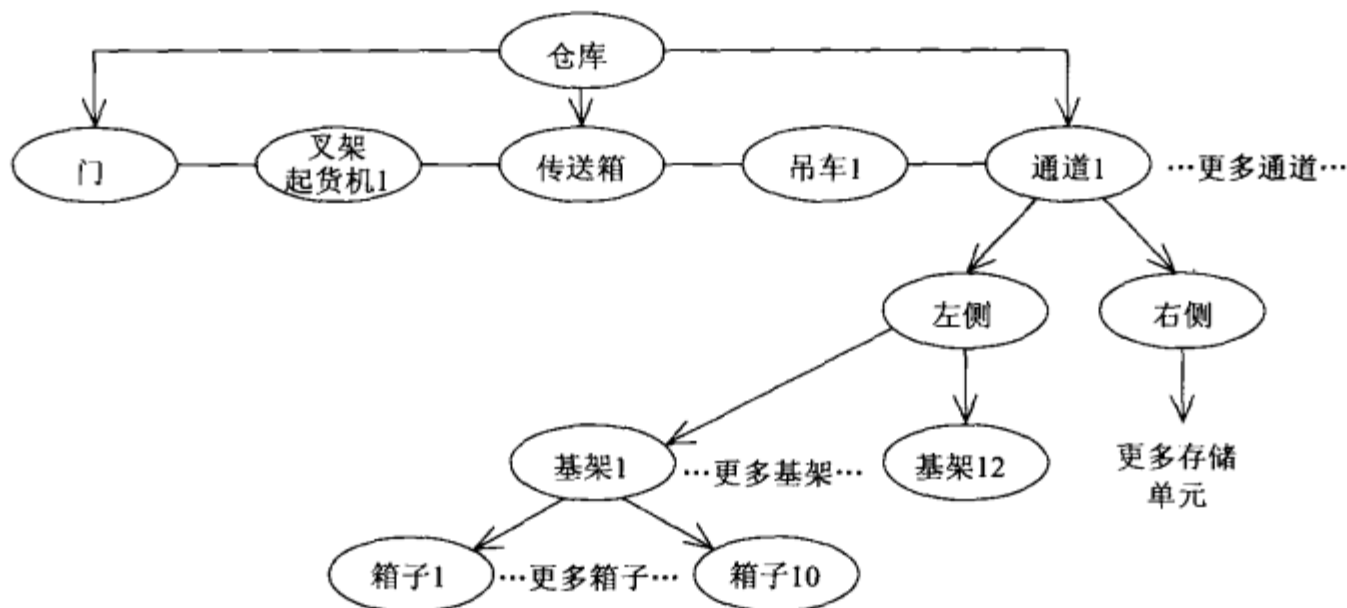


图 7-3

Composite模式还能满足我们扩展和重新组织仓库结构的要求，通过将相应的新存储单元和传送设施类集成到该层次结构中，实现新的存储类型和传送设施类型。对现有存储单元和传送设施的更改或移除也不会影响结构中的其他类。

最后，Composite模式可以很容易地映射成Half-Object plus Protocol (188) 的分布式策略，以实现联盟的（federated）仓库拓扑领域对象。所有半对象的根组合元素通过适当的协议连接起来。从逻辑角度来看，这允许仓库拓扑的客户端访问整个仓库结构，而不需要关心它是如何在网络上分布的。类似地，具体的Composite结构可以映射到Active Object (212) 并发模型——不同的子树分配给不同的Active Object。例如，可以根据现实世界中相应的并发性为每个仓库站点、每个建筑、每个通道或通道的每一侧提供一个Active Object。

将来我们可以在一个Active Object内部提供并发能力。例如，如果某个Active Object代表一个仓库站点，它可以包含一个线程池，其中每个线程包含该站点中一个不同仓库建筑的Composite层次结构。

7.3 存储结构导航

仓库管理流程控制系统的关键操作需求之一就是性能，在实现了仓库拓扑领域对象的环境中，解决性能问题的核心方法是Half-Object plus Protocol (188) 分布式策略和Active Object (212) 并发模型。但是性能还和仓库表现采用的Composite (185) 方式有关——我们访问层次结构中的特定存储单元越快，对相应请求的服务和处理就越快，系统的整体请求吞吐量就越大。

然而，绝大多数Composite方式只是简单提供了严格的自上而下的导航方式：访问存储元素某个特定实现的请求总是要从层次结构的顶层开始，沿着特定路径直到找到所指定的接收者。层次结构越深、在路径中穿越进程和线程边界的步骤越多，那么这一方案就越不可行。尤其是当一系列的并发请求都“落在”该结构的同一区域，比如通道的同一侧或同一基架时更容易引发很多问题，而这在处理特定物品的大型发货或收货订单时很常见。这种情况下如果从当前存储单元——通常是一个箱子——沿Composite层次结构向上访问会更好，往上一层或上上一层存储单元——比如包含箱子的通道，再往下一层存储单元——比如通道中的另一个箱子。与严格的自上而下的请求传递相比，这种策略可以节省很多步骤，而且在很多情况下不需要穿越线程或进程边界。

我们怎样才能采用Composite方式的存储层次结构中支持双向导航，从而可以在该结构中从任意存储单元最有效地访问其他存储单元呢？

将层次结构中的所有存储单元连接到它们的上一层的存储单元，构成从所有叶子到根元素的Chain of Responsibility (257)。通过Composite模式完成自上而下的导航，或通过Chain of Responsibility完成自下而上的导航。

将Chain of Responsibility集成进我们的设计中来表现物理仓库结构是很直观的：将Storage类扩展，与自身构成parent关系。

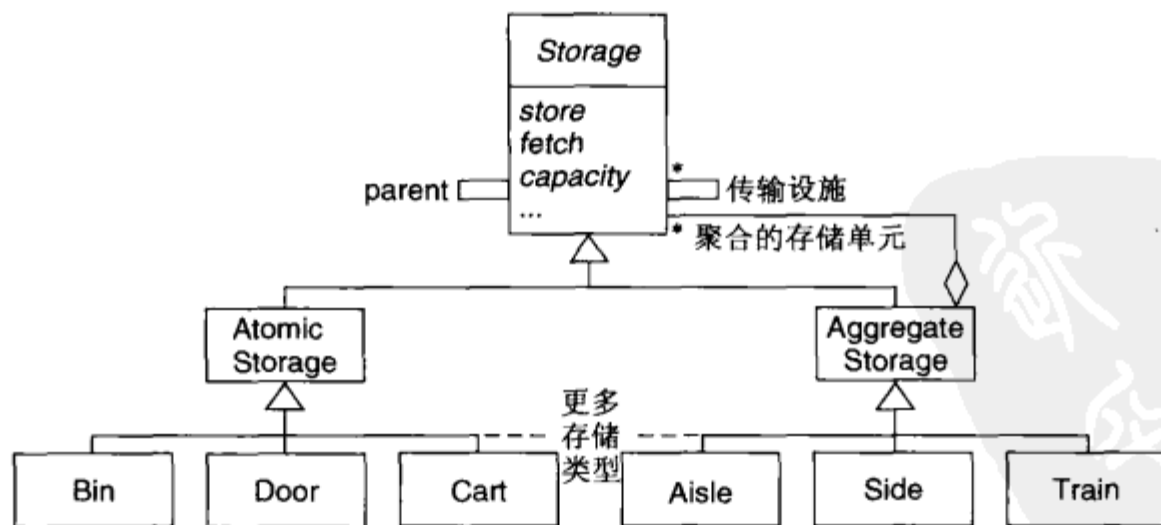


图 7-4

在具体的仓库配置中，仓库层次结构中的代表“真正的”存储部件的单元和相关的聚合存储部件连接起来可以支持双向导航。

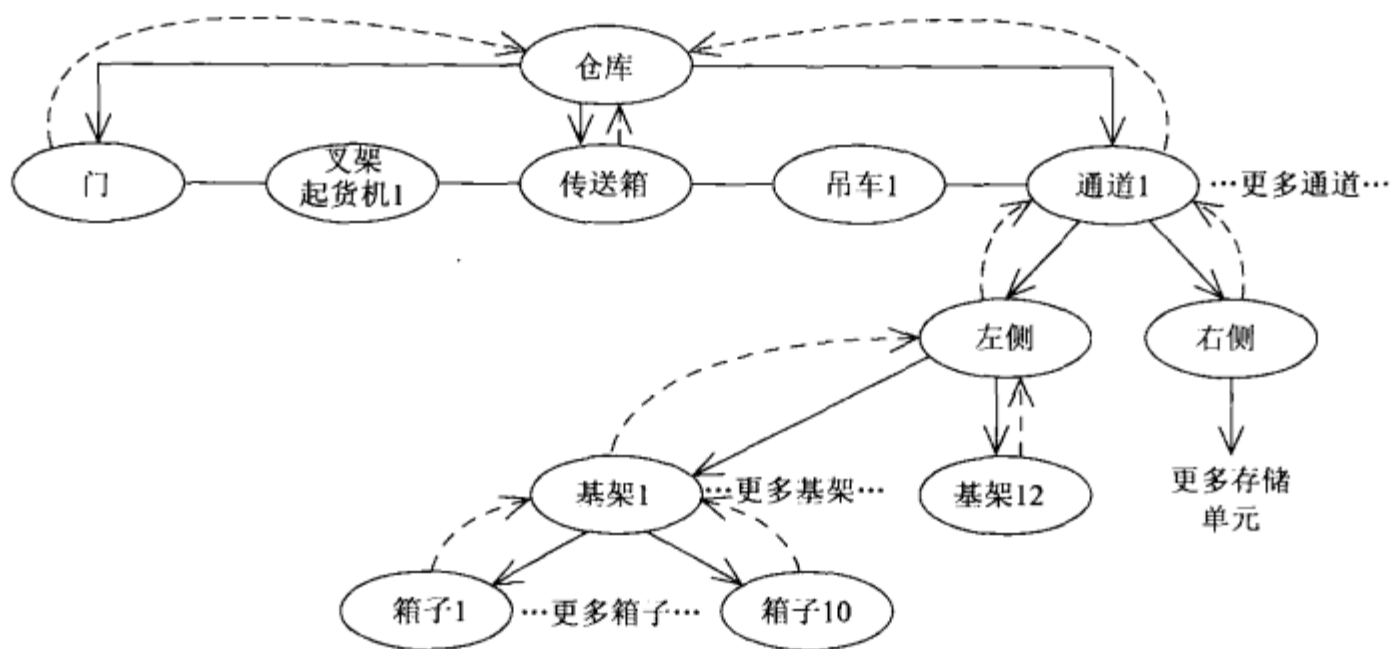


图 7-5

在采用Composite模式的层次结构中存储单元之间的双向链接确保了访问每个存储单元只需要最小的步数、穿越最少的线程边界。

7.4 存储属性建模

仓库中的每个存储单元、传输设施以及物品都与特定属性相关。这些属性被称为存储组织标准（Storage Organization Criteria, SOC），并决定了可以存储在特定存储单元中的物品类型，或可以由特定传输设施传送的物品类型。存储组织标准的例子有吞吐量类、有害类以及温度类。对每一个标准都有特定的、可枚举的允许值，如对吞吐量类来说有快速、中等和低速几种。仓库管理流程控制系统中的领域功能使用这些信息来决定要存放物品的目标位置、从当前位置到目标位置的中间步骤，以及每一步骤用来传输物品的传输设施。例如，只存储一小段时间就快速取出的物品属于吞吐量类的快速类，这时应当将物品只存放在快速类的存储单元中。类似地，具有某种有害性质——如易燃性的物品，则应当由适合传输易燃物的传输设施来传送。

某一特定存储单元、传输设施以及物品的存储组织标准通常很少随时间变化，除非是对仓库进行现代化改造。在正常的仓库运行中它们通常都是固定不变的，一旦确定了特定存储单元、传输设施或物品，则存储组织标准就只能由相应的管理功能来显式地调整。

不过，在仓库管理流程控制系统的具体安装中，维护这些数量庞大的存储组织标准就产生了一个问题。将每一个存储单元、传输设施和带有私有的相关存储组织标准集合的物品联系起来需要大量的内存，比如，有的仓库可能会有超过一百万个箱子。此外，因为许多仓库拓扑元件共享相同的存储组织标准，这会“浪费”很多内存。但是，为了确保仓库操作正确，又必须使得它们中的每一个都与一个良好定义的存储组织标准集合联系起来。

我们怎样才能为仓库中的每个存储单元、传输设施以及物品提供相应的存储组织标准列表，而避免消耗巨大的内存呢？

将存储组织标准采用 Immutable Value (231) 来实现，从而在仓库拓扑的多个元件之间共享。

每个存储单元、传输设施和物品都维护着一份不变的存储组织标准集，这是在具体的仓库配置时指定的。在我们的设计中这个集合由 Composite (185) 结构的根类——Storage 类维护。

一个 Immutable Value (231) 可以由不同的存储单元、传输设施以及物品共享，甚至多个控制线程也可以共享同一个 Immutable Value。此外，仓库管理流程控制系统中的领域功能不能修改 Immutable Value，而只能通过 Factory Method (313) 由相应的管理功能来显式改变。

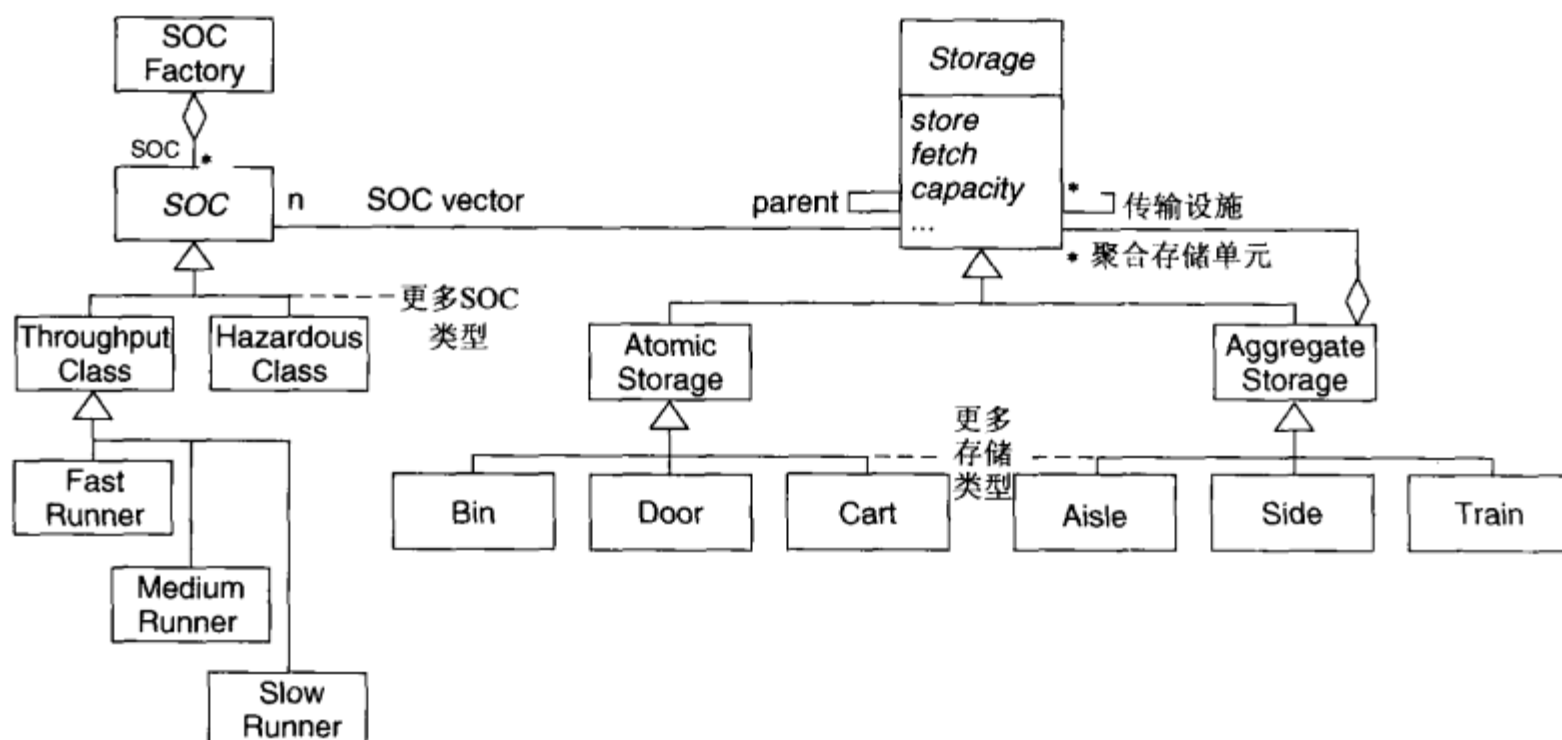


图 7-6

7.5 不同的存储单元行为

为了表现具体仓库的拓扑结构，Composite (185) 设计提供了一个公共的显式接口，可以针对不同类型的存储单元和传输设施采用不同的实现。例如，将某一物品存到通道中意味着选择了通道中合适的一侧和基架，然后再选择指定基架上的特定箱子，最后将物品放入箱子则意味着将物品的标识与箱子联系起来。

公共接口中特定方法的具体行为不仅会因不同的存储单元类型而不同，还会因为同一存储类型的不同实例而不同。例如，在一个几乎是空的通道中，物品按“层”从底往上存放在所有的基架上，而当通道中的东西超过一定界限时，物品按“堆”填满一个又一个基架。前一种存储策略就吞吐量来说优化程度不足，因为传输设施（如吊车）需要移动更长的距离才能存取特定数量的物品，但它可以保持通道重心的平衡，进而确保基架的静态稳定性。后一种策略将物品集中存放起来，可以保证存取物品只需要移动很短距离，但只有当堆积物品时不会危及通道稳定性时，才能安全工作。理想情况下，通道存放策略可以根据当前的状态随时间而改变，这样就可以在任意

时刻使用最合适的存放策略。

我们怎样才能能在Composite结构中支持类方法的多个实现并在运行时改变当前“活动的”方法实现呢？

为所有可能变化的方法提供Strategy (266)，使用多态来配置和重新配置Composite安排的存储单元以使用合适的方法实现。

就Strategy模式来说，我们当前设计中的Storage类扮演上下文（context）的角色。Composite结构类实例中行为可变的方法被指定为StorageStrategy类的Explicit Interface (163)，如取（fetch）和存（store）方法，由子类来实现，如StoreInLayers和StoreInPiles。Storage类提供了一个方法来配置Composite方式中特定实例的具体存储策略。

Strategy模式支持在公共的组合接口中提供实例相关的方法实现，并能在运行时改变。后者可以通过调用Storage类中的配置方法来显式触发，或通过Composite结构中类的某些内部逻辑来实现。

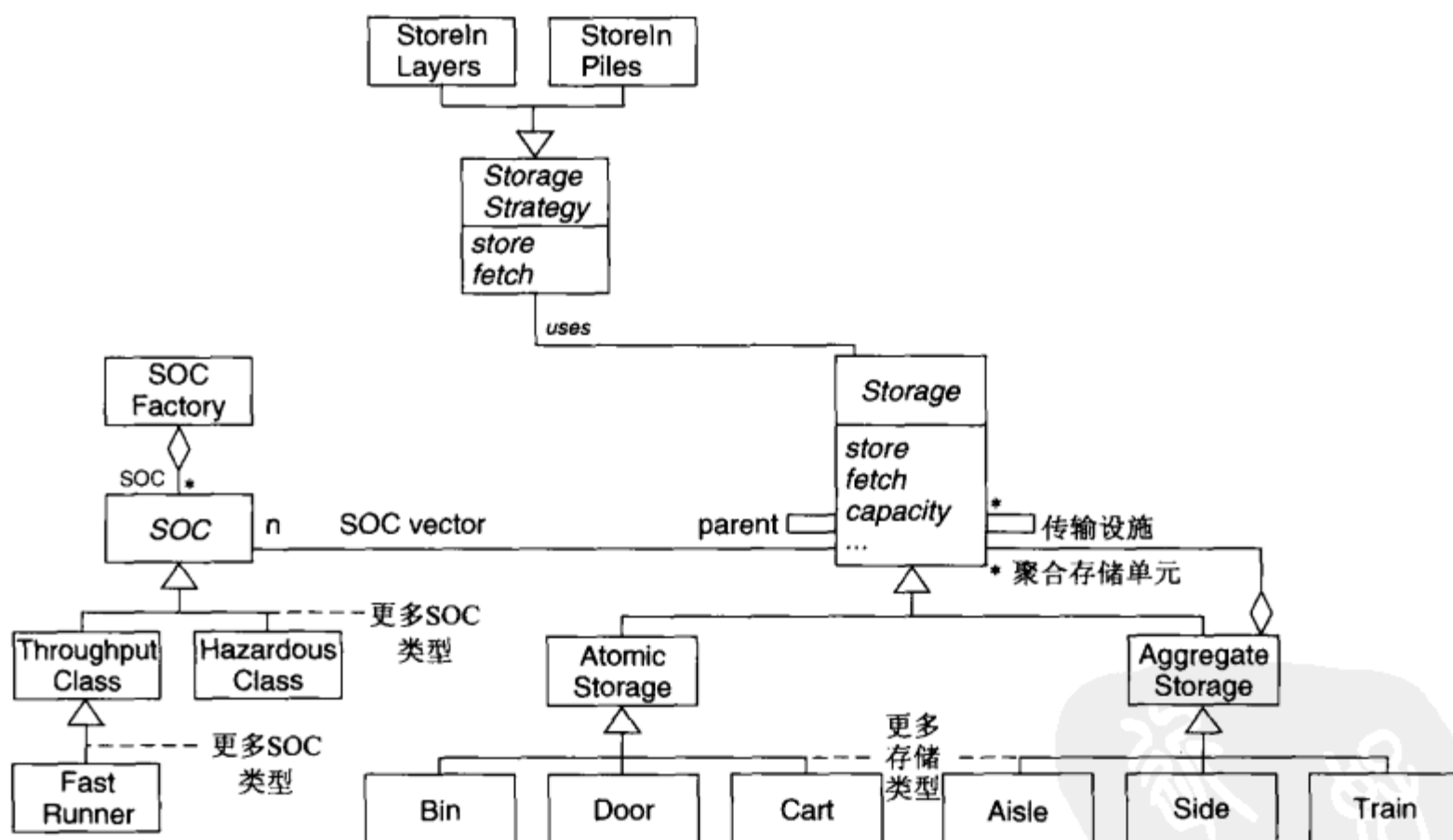


图 7-7

除了Strategy模式之外，Template Method (265) 也可以提供一个相同的可行方案以支持类的不同功能行为。Strategy模式通过委托来解决该问题，而Template Method通过继承来实现。不过，根据“斯堪的纳维亚面向对象学派”的说法，仓库管理流程控制系统的架构设计原则应该是：如果类之间不是纯粹的结构性的关系，我们应当尽量采用委托的方式而不是继承。因此，在设计仓库拓扑领域对象中使用Strategy模式更好，而不是Template Method。

7.6 实现全局功能

有些仓库拓扑管理功能，比如收集统计信息或重组，是针对整个仓库拓扑的。这些功能遍历所有或大量的存储单元，并进行存储单元相关的操作。例如，重新组织仓库，与硬盘的碎片整理相类似，随着时间流逝，大量的存取操作会使得仓库中的存储单元变得零碎，以至于基架上只是零星地填充着少量物品，同时某一类物品却分散在整个仓库。这会导致处理特定数量的物品时需要更多的传输时间，从而降低仓库的吞吐量。通常的仓库重组通过重新存储物品来“清理”仓库，这使得连续的仓库操作得以高效执行。

然而，目前基于Composite (185) 的设计只支持那些在仓库拓扑中的一个或多个存储单元上进行本地操作的功能，如存、取或传输物品，而像重新组织仓库这样需要全面操作大量存储单元的功能很难用这种方式进行模块化处理，它必须被分解为由多个类实现的不同部分。因为其逻辑和控制流分散在多个设计和代码单元中，进而导致这些函数难以理解与维护。此外，这些类与Composite结构的耦合也越来越紧密，改变一个类中的全局功能可能需要在其他类中做相应的改变。因为这种耦合只是隐含的而没有显式表现出来，同样违背了结构良好软件的基本设计原则，使得设计看起来比实际上的耦合度要低。

我们怎样才能支持用严格的模块化方式实现全局拓扑管理功能，而不破坏当前基于Composite设计的结构特性呢？

用Visitor (261) 模式实现全局性的拓扑管理功能，以遍历具体仓库拓扑的所有存储单元并在选定的单元上执行（本地）操作。

将Visitor模式集成进我们的设计，与集成实现类特定行为的Strategy模式非常相似。GlobalOperation类声明了对Composite设计中每个具体存储单元类的visit方法。它相当于所有全局性仓库拓扑管理函数的Explicit Interface (163)，由实现visit方法的相应子类来实现，并扩展Storage类以增加一个accept方法来接受Visitor，而Composite结构中的具体存储单元类通过回调相应Visitor的visit方法来实现这个accept方法。

使用Visitor模式实现操作仓库拓扑中大部分元素的方法，有助于保持设计的模块化结构和清晰——不同的关注点彼此分离，如针对局部存储单元和针对全局存储单元的功能。考虑到性能优化，Visitor可以在它们自己的控制线程中运行，这样其他操作仍然可以在Visitor工作的时候同时运行。

7.7 遍历仓库拓扑

许多仓库管理的操作，为了正确执行需要遍历整个仓库拓扑。例如，收集统计信息需要访问仓库中的每一个存储单元，而存取一个物品需要找到一个合适的存储单元。

然而，将遍历性的功能集成进相应的仓库管理操作中，会使其依赖于表示仓库拓扑的具体Composite (185) 结构。类似地，将遍历性的功能集成进Composite结构自身，则需要复杂的基础设施以支持多个同时的遍历。

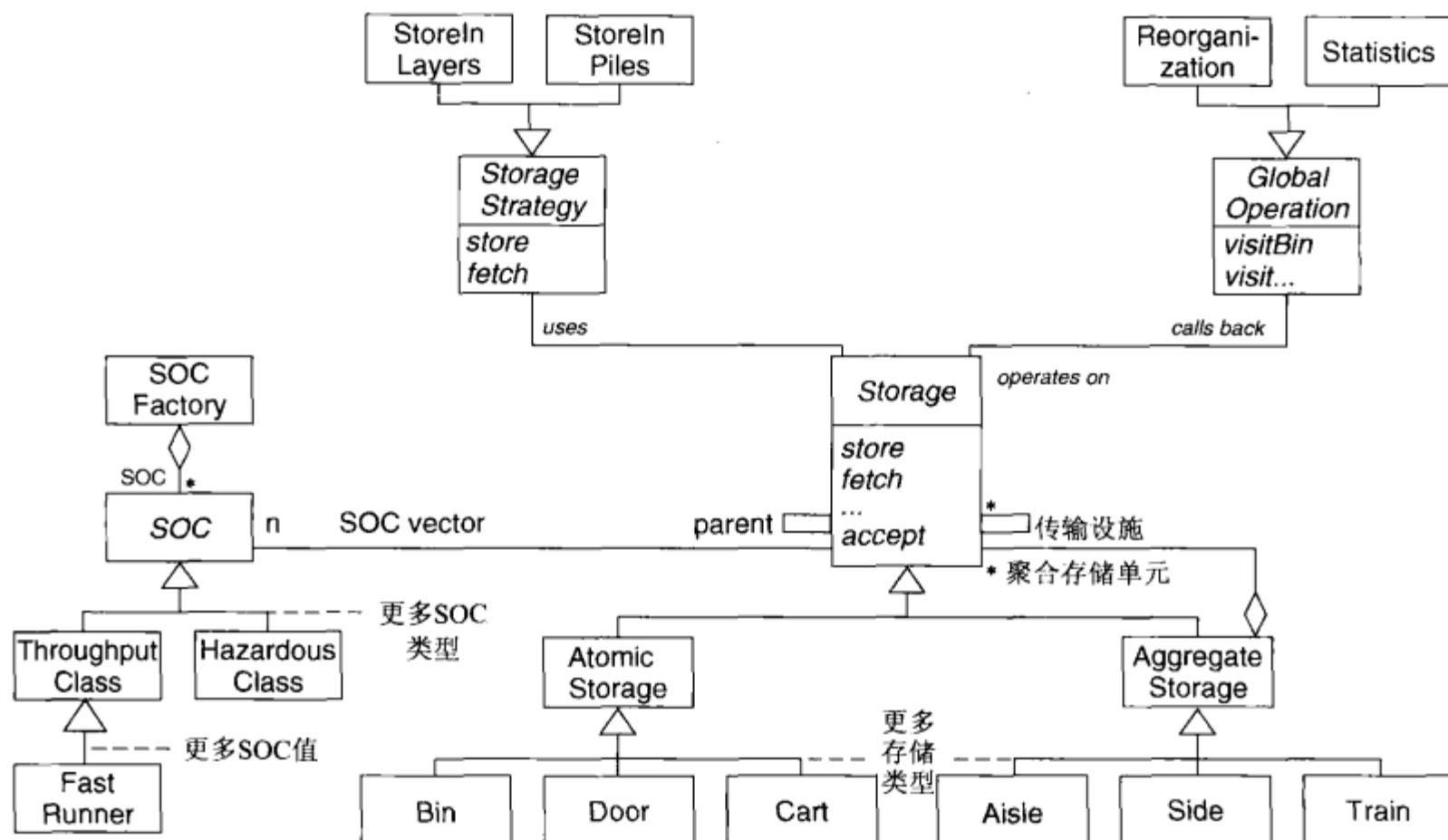


图 7-8

我们怎样才能支持对仓库拓扑中存储单元的连续访问，而不需要暴露其内部结构呢？

使用Iterator (173) 将遍历策略与仓库拓扑表现和需要访问的操作分离开，让具体的Iterator实例维护特定遍历过程的状态。

Iterator类定义了Explicit Interface (163)，例如一个叫next的Combined Method (172) 用来访问和遍历仓库拓扑中的存储单元。具体的子类来实现特定的遍历策略，如只遍历原子存储单元或只遍历传输设施等。客户端可以通过调用Storage类提供的Factory Method (313) 获取存储结构的Iterator。

将遍历策略提取到单独的Iterator中，不仅使得我们设计中的领域功能和核心数据结构不受“工具方面” (utility aspects) 的影响，如特定方式的数据访问等，还能在不影响设计中的其他类的情况下增加、删除和修改遍历策略，从而能够（动态）配置和重新配置领域功能以提供不同的遍历策略。

图7-9描述了在仓库拓扑领域对象中是如何使用Iterator的。

第一眼看上去Visitor模式 (261) 似乎可以代替Iterator模式。像Iterator一样，Visitor允许我们分离仓库拓扑表现和遍历功能，并支持多个并发性的遍历。然而，Visitor通过双向指派来实现这一功能。这种机制在遍历并完成与所访问存储单元相关领域功能时比较合适，如重新组织，但是对于只是一个接一个的访问仓库拓扑中元素的情况来说就过于复杂了。

使Composite (185) 结构中的特定Storage类可以在这些事件发生时回调它们，同时仓库拓扑的当前运行状态信息会通过回调传递给观察者。

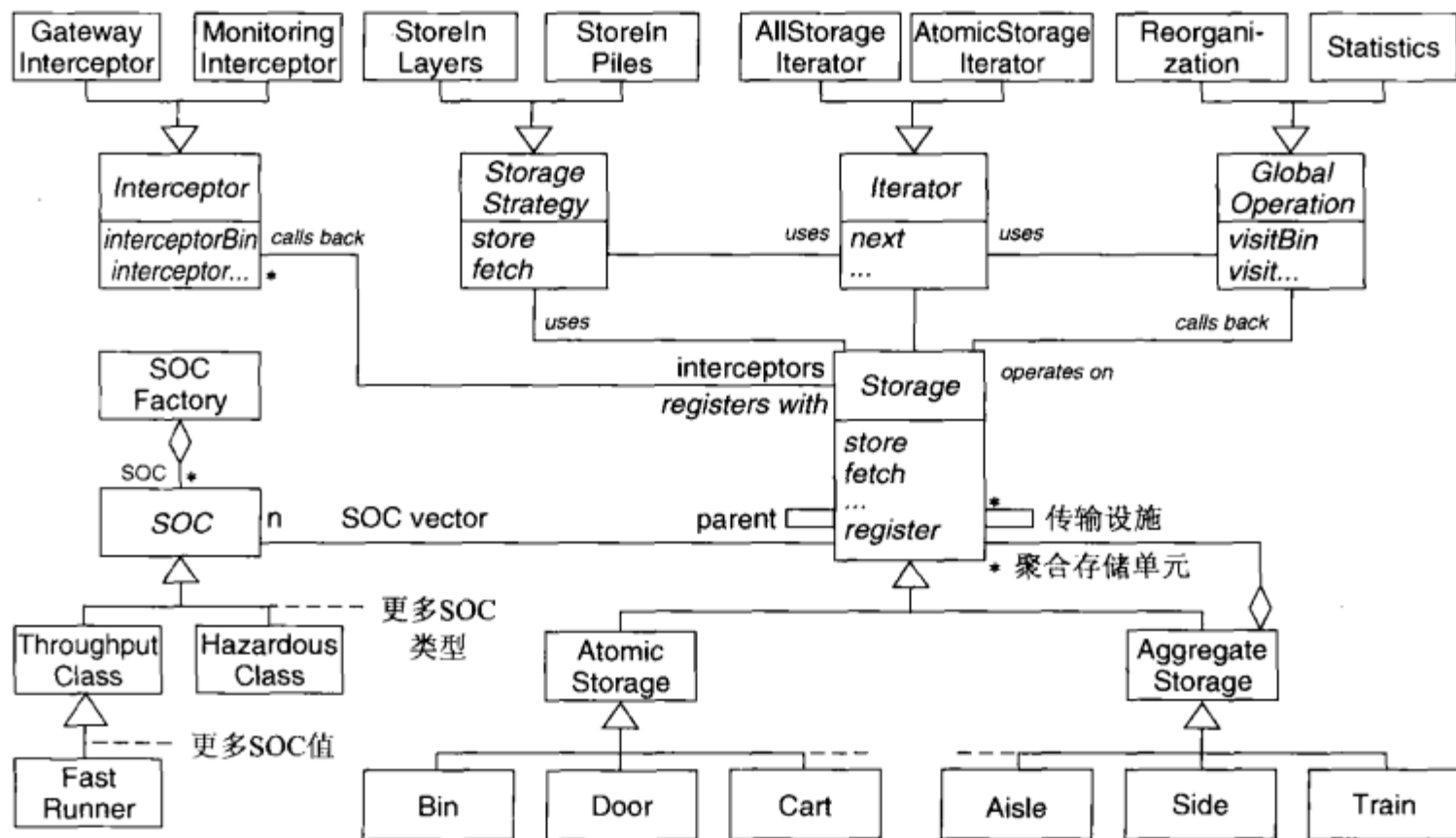


图 7-10

为了提供可插拔的Interceptor框架，仓库拓扑领域对象的核心设计对客户和系统专属的已有功能的扩展是开放的，而同时可以维持原来简练稳定的结构，因而非受控的改变是封闭的。

附加控制流扩展还可以通过Decorator (262) 设计来实现，Decorator模式通过封装已有类来提供额外功能或行为。然而，Decorator模式只能给类增加新的方法，或对已有方法提供预处理或后处理功能，而不能对给定方法提供额外的控制流。因此我们在设计仓库拓扑领域对象时更愿意采用Interceptor而不是Decorator模式。

7.9 连接数据库

正如第4章仓库管理流程控制所描述的，仓库拓扑永久性地维护在数据库中。Database Access Layer (318) 确保仓库管理流程控制系统的架构与具体的数据库接口及相应的数据库范式无关。然而，性能和吞吐量的需求决定了在数据库中操作整个仓库拓扑是不现实的，我们还需要将仓库拓扑存放在内存中，以便能高效地完成相应的操作。另一方面，在内存中保留一份仓库拓扑的完整复本也是不现实的，对于大的仓库来说，要维护整个拓扑所需要的内存会远远超过可用内存的大小，即使仓库拓扑分布在多台主机上也是一样的。

我们怎样才能确保无论仓库拓扑操作访问哪个存储单元，它都可以从主存中得到呢？

为仓库拓扑中所有原子存储单元提供Virtual Proxy (294)。不管什么时候需要通过Virtual

Proxy访问原子存储单元，都会在相应操作执行之前将数据从数据库载入到内存中。一旦存储单元已经在主存中，则会将其保持在内存中以便后续访问能高效执行。

提供基于代理的方案来访问原子存储单元增加了仓库拓扑领域对象的结构复杂性，然而，一旦数据被载入到内存中，系统性能的提升远远大于设计上的付出。从性能优化的角度来说，如果原子存储单元的数据已经在内存，还可以绕过代理直接访问。为聚合存储单元提供Virtual Proxy在技术上是可行的，但是因为对聚合存储单元的访问比较频繁，因此将数据一直保留在内存中会比在需要时载入效率更高。此外，和原子存储单元的数量相比，在具体的仓库配置中聚合存储单元的数量要少得多，因此不会浪费过多的内存。

在原子存储单元中支持Virtual Proxy非常简单：从Storage类派生出StorageProxy类并建立到具体AtomicStorage的关联，可以通过主键实现。当需要访问代理的时候，通过主键载入相应的存储单元数据并执行相关的操作。

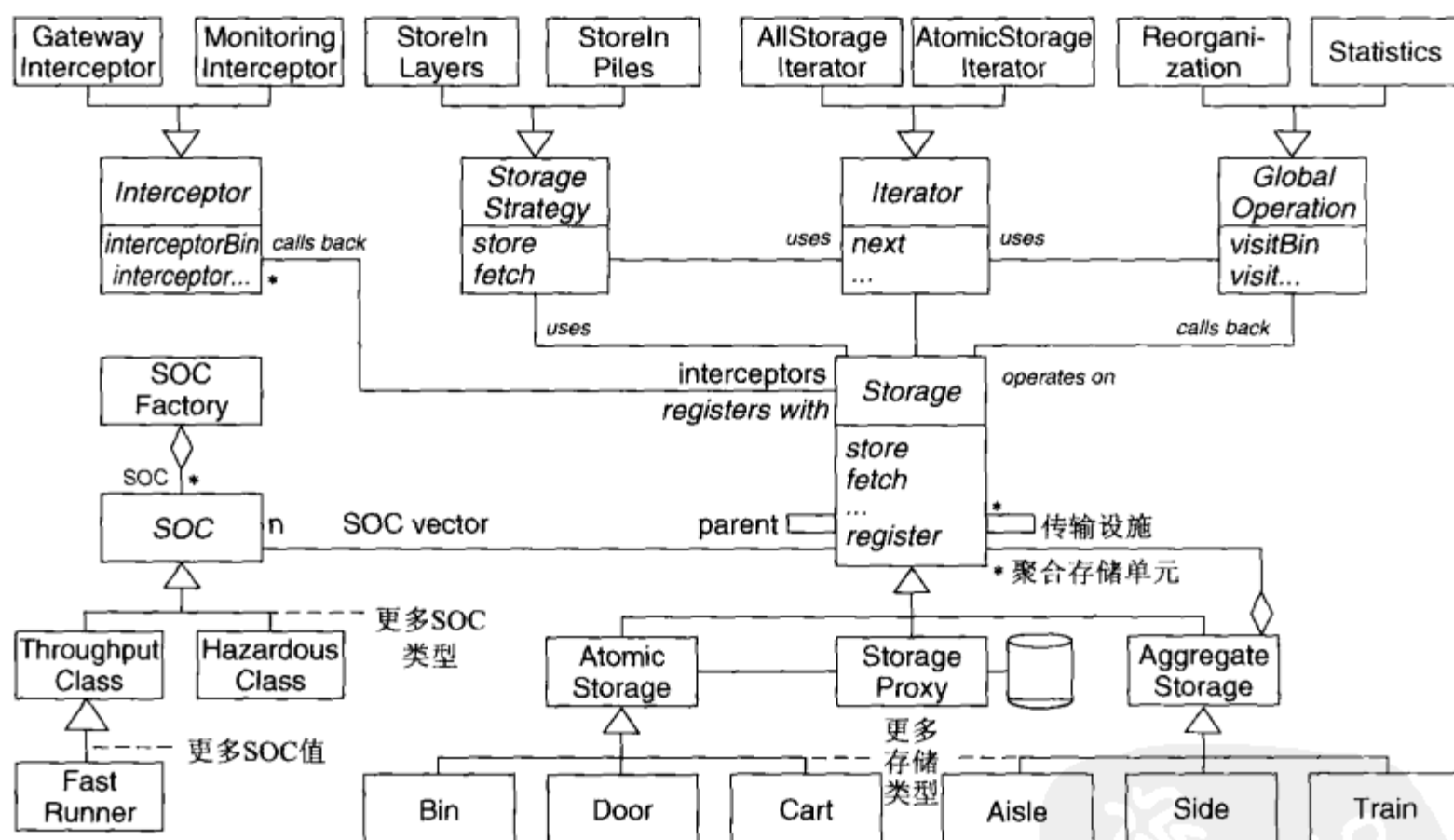


图 7-11

7.10 维护内存中的存储单元数据

尽管在仓库拓扑设计中引入代理能达到性能和吞吐量方面的目标，它同时也引入了一个明显的缺点。通过代理载入到内存中的存储数据不会在操作完成时清除，相反它一直保留在内存中以便后续操作的高效执行。这样一段时间之后可能会占用所有可用内存，从而不能从数据库中再载入更多数据。

修改代理的行为，从而在操作完成时删除相关数据并不是该问题的合理解决方案，我们会面对和原先讨论最初代理实现时相同的性能问题！

我们怎样才能保持原有的Virtual Proxy (294) 设计来表现原子存储单元，又能避免内存耗尽呢？

在内存中提供固定大小的Resource Cache (299) 以维护仓库结构。如果资源缓存满了，则删除不再使用的那些存储数据，从而可以安全地将新的存储数据通过它们的代理载入到内存中。

在缓存中维护仓库存储数据，以将资源管理引入到我们仓库拓扑设计中。它使得我们能降低基于代理访问原子存储单元所带来的缺陷，内存消耗从原先无法控制到可接受的最小值，同时保留原有的性能优势。

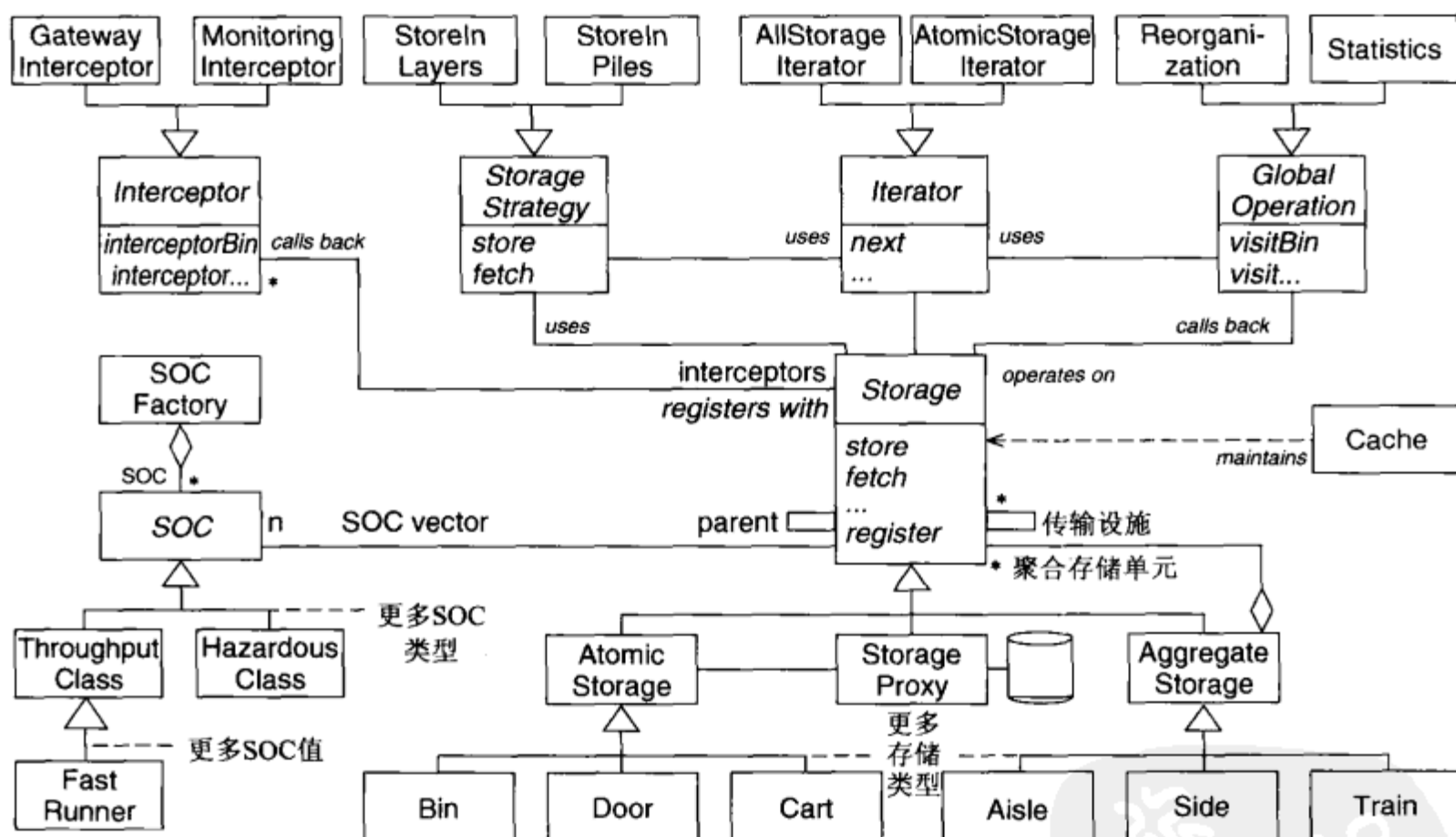


图 7-12

7.11 配置仓库拓扑

我们为仓库拓扑领域对象建立的设计提供了多样的配置选项：Interceptor、Visitor、策略以及 Iterator，以便灵活适应仓库特定的行为，而 Composite (185) 结构则有助于集成仓库特有的存储结构。

然而，配置一个特定的仓库拓扑领域对象实例并不那么容易。不同的配置选项之间可能会在语义上相互依赖——例如，特定的 Visitor 可能需要特定的 Iterator 才能正确工作——而且配置选项也常常必须按照定义好的顺序安装好。采用“人工”方式通过适当接口来显式配置每个选项可能

在理论上是可行的，但是却可能会为那些微妙的配置错误打开方便之门，而且这些错误往往到仓库拓扑在系统操作中出错时才会被注意到。

我们怎样才能简单并正确地配置仓库拓扑Domain Object (121)?

使用仓库拓扑Domain Object内部的Builder (312) 来检查一套特定配置选项以保证一致性，即创建每个配置选项，并将所有选项按照适当的顺序集成到适当的地方。

通过仓库拓扑领域对象的显式接口中适当的配置方法，系统的中心Component Configurator (289) 基础设施将Storage类、Interceptor、Visitor、Strategy以及Iterator的集合传递给它内部的Builder。由ConfigurationDirector类来控制配置过程：它首先检测接收到的配置项的一致性，如果正确的话再指示ConfigurationBuilder类按照正确的顺序为每个元件安装配置项。ConfigurationBuilder类首先创建相应的配置元素，然后再将其集成到仓库拓扑结构的适当位置。对Builder来说，所配置的具体Storage类、Interceptor、Visitor、Strategy，以及Iterator扮演产品的角色。

图7-13描绘了简化后的仓库拓扑领域对象示例配置。

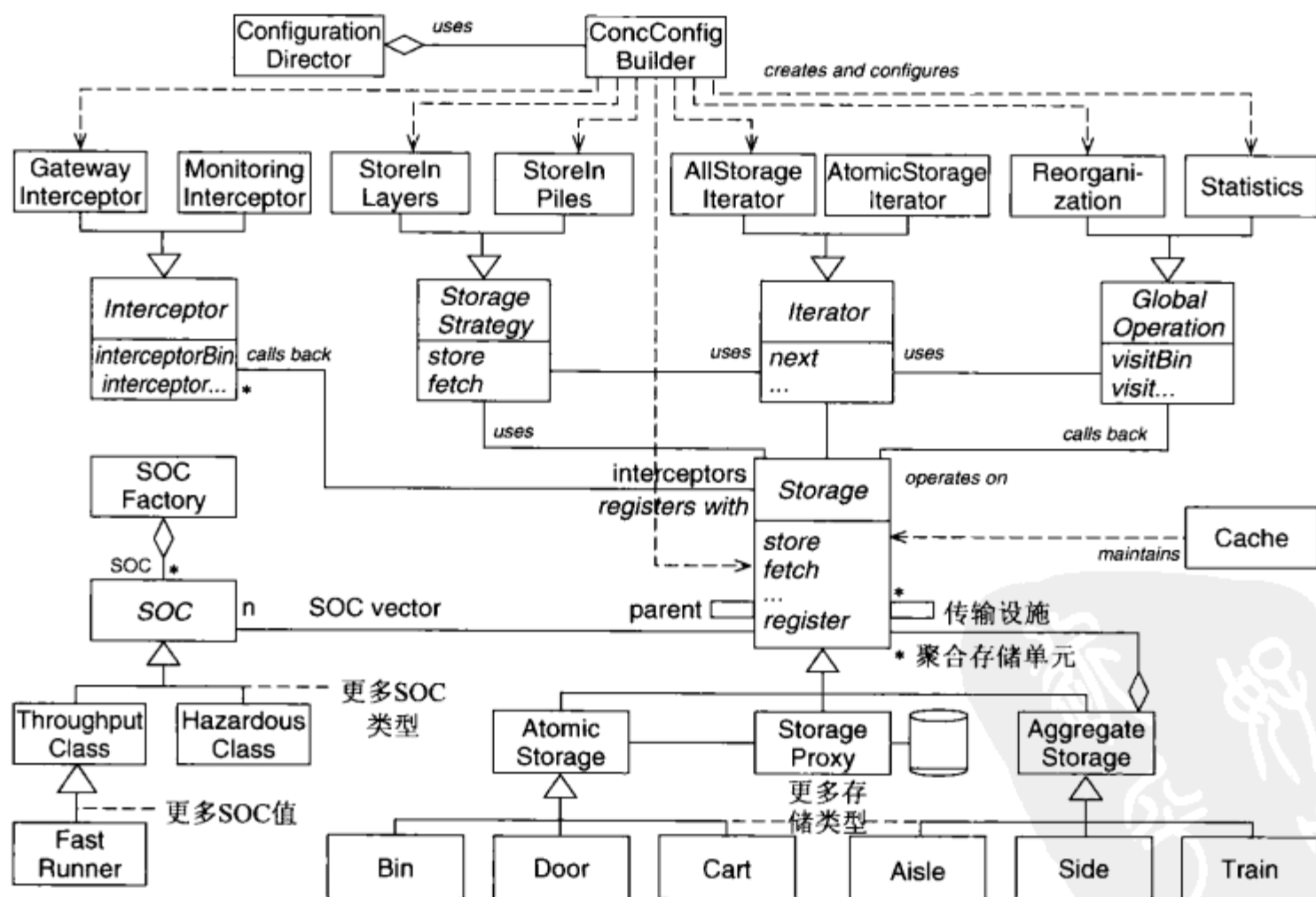


图 7-13

使用Builder方式来配置仓库拓扑领域对象的具体实例有利于提高系统的简单性、有效性和正确性。简单性是因为仓库拓扑及其领域功能都不需要关心配置的正确性。相反，为此专门提供了

一个单独的实体类：**Builder**。高效性是因为**Builder**是仓库拓扑领域对象的内部类，其配置过程的实现能利用和访问仓库拓扑具体设计和实现以及相关知识。正确性是因为**Builder**实现了经过显式编码和测试的配置策略，它在合适的位置按合适的顺序集成了不同的配置选项。

7.12 细述显式接口

仓库管理流程控制系统的基线架构指明每个领域对象必须提供能远程访问其功能的显式接口。

然而，像仓库拓扑这样的领域对象为客户提供了范围很广的不同接口，而不同客户可能只需要使用其部分功能。为整个仓库拓扑提供单一的显式接口可能会在系统的领域对象之间引入不希望的隐含依赖。例如，如果某个客户领域对象并不使用仓库拓扑提供的所有功能，而它没有使用的函数签名发生了变化，那么即使它自身对仓库拓扑的使用没有发生变化，该客户端也必须重新编译和链接。类似地，如果仓库拓扑接口扩展了新的功能，即使客户端不使用也必须重新编译和链接。理想情况下，仓库拓扑的客户端只在它所使用的仓库拓扑发生部分改变时才需要相应改变，比如用到了新增功能或现有功能的使用发生了变化。

我们怎样才能确保Explicit Interface (163) 的变化——不管是更改现有函数签名还是提供新函数——只影响对该接口变化有兴趣的客户端呢？

将Explicit Interface划分为角色相关的Extension Interfaces(165)，为每个角色分配一个或多个扩展接口。永远不去改变已公开的扩展接口：如果一个Domain Object (121) 必须提供额外的角色支持，引入相应的新扩展接口，如果现有的角色改变了，也为这个改变了的角色提供一个额外的扩展接口，但是仍然支持原有的扩展接口。

在仓库拓扑的Extension Interface中，RootInterface提供访问任意扩展接口以及在所有提供的扩展接口之间导航的功能。从RootInterface派生出具体的角色专属的扩展接口，如Configuration、Routing和Statistics接口。客户端可以通过TopologyInterfaceFactory获取最初的扩展接口。一旦客户端可以访问某个特定的扩展接口，它可以通过接口的导航功能来访问仓库拓扑提供的任意其他扩展接口，只要它拥有相应的访问权限即可。

图7-14概述了仓库拓扑的Extension Interface设计。

为仓库拓扑提供角色专属的扩展接口，极大地提高了它在可控改进方面的开放性。对领域对象显式接口和封装实现的基本划分，加强了客户端在实现改变时的稳定性。扩展接口增强了这一点，甚至在仓库拓扑对外公开的契约发生特定变化和扩展时还能够保持客户端的稳定性，特别是当客户端使用的扩展接口不受仓库拓扑新增扩展接口的影响时，或其他未使用的扩展接口发生变化时，客户端都不会受到影响。如果客户端对其所使用接口的演化“不感兴趣”，同样可以不受影响，因为系统仍然支持旧的接口。因此，扩展接口完善并补充了我们对仓库拓扑封装实现的灵活设计。

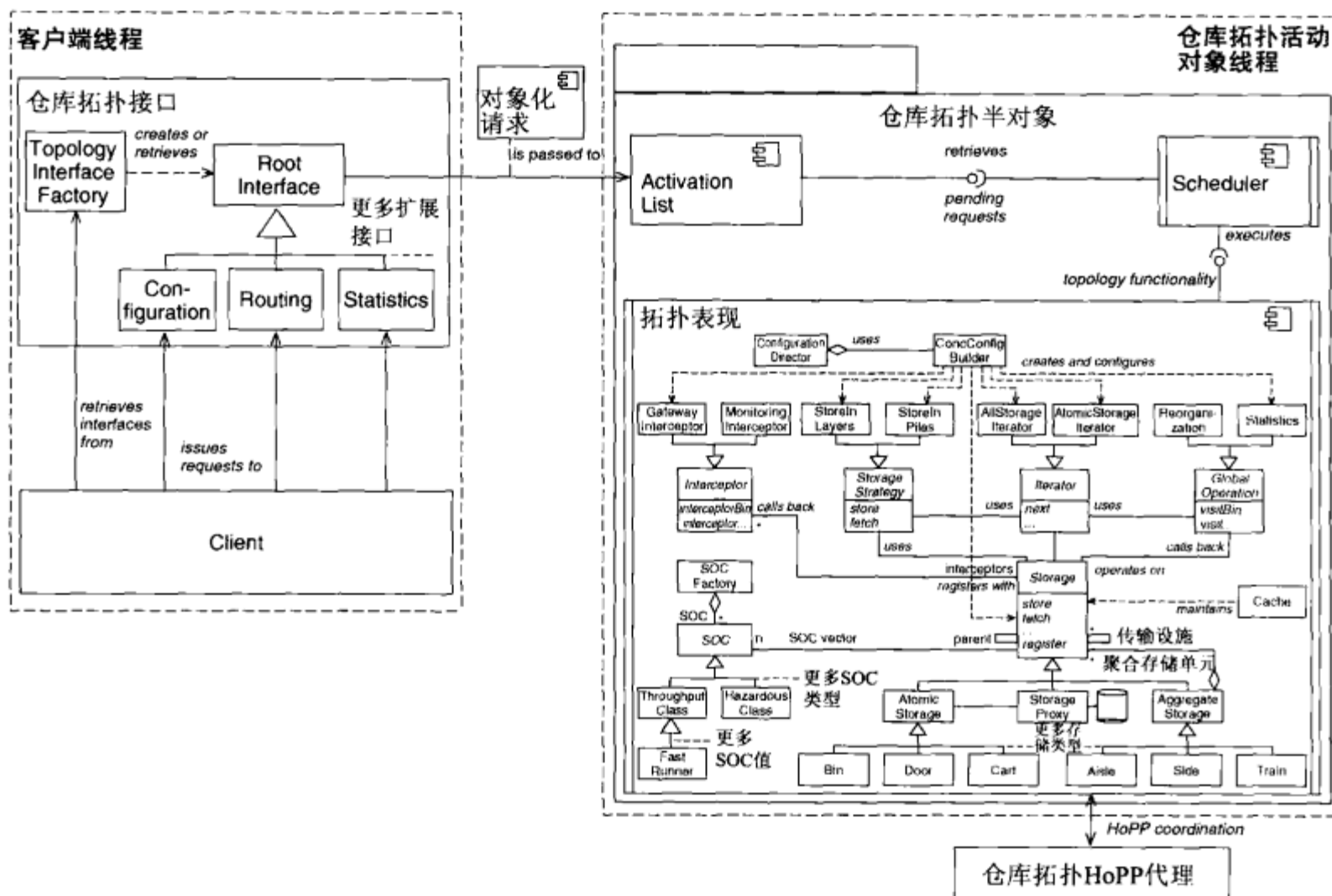


图 7-14

7.13 仓库拓扑总结

从总体上观察一下我们在设计仓库拓扑Domain Object (121) 中应用的模式序列就会发现，即使该设计是针对仓库管理领域的，仍然可以从我们的分布式计算通用模式语言中得到创建该序列的具体指导。

前3个模式——Composite (185)、Chain of Responsibility (257) 和Immutable Value (231) ——构成了仓库拓扑Encapsulated Implementation (181) 的战略核心，以及用来表示和容纳具体仓库物理存储单元和传输设施的空间分布和功能行为的中心结构。

4个模式展现了支持(战术上的)行为变化所需的稳定设计[Gam97]。Strategy(266)支持特定存储单元或传输设施在行为上的不同; Visitor(261)允许在多个存储单元和传输设施上进行操作,它还支持仓库拓扑扩展,以支持在原始设计实现中没考虑到的功能; Iterators(173)允许Strategy和Visitor按不同的遍历策略来遍历仓库结构;最后Interceptor(260)允许仓库拓扑核心控制流以受控的方式扩展,以支持客户特有或系统特有的额外功能。

接下来的两个模式解决仓库拓扑领域对象运行质量的重要问题——资源。Virtual Proxy (294) 允许我们只在主存中保留仓库拓扑的核心结构, 特定存储单元和传输设施的具体数据保留在数据

库中，并只在需要访问它们时才被载入到内存中。Resource Cache (299) 缓解了Virtual Proxy的主要性能瓶颈，载入到主存中的数据不应当在使用后立即删除，而是保留在内存中以便后续快速访问，但是这些数据不应该无节制地吞噬内存。

Builder (312) 模式完善了仓库拓扑的设计，它提供集中机制以正确可控的方式配置仓库拓扑Domain Object实例。

最后，Extension Interface (165) 将仓库拓扑的Explicit Interface (163) 划分成多个基于角色的契约，每个契约都可以独立地访问和改变。

表7-1总结了帮助我们创建仓库拓扑领域对象的模式序列，以及序列中每个模式所解决的设计挑战。

表 7-1

模 式	挑 战
Composite	仓库存储结构建模
Chain of Responsibility	向上浏览Composite结构
Immutable Value	共享状态信息
Strategy	支持存储单元相关的行为
Visitor	提供全局性的仓库功能
Iterator	遍历仓库结构
Interceptor	提供控制流扩展
Virtual Proxy	访问持久化数据
Resource Cache	维护内存中数据
Builder	配置仓库拓扑Domain Object
Extension Interface	提供基于角色的仓库拓扑接口

需要注意的重要一点是，尽管仓库拓扑领域对象看起来可以为其他必须维护拓扑信息的系统“重用”，其实它本身是一个由特定需求驱动的独特设计。例如，如果整个仓库拓扑能维护在数据库中而不影响效率和性能，可能就不再需要专门的仓库拓扑领域对象。通过恰当的设计，像第5章基线架构中描述的采用Database Access Layer (318) 设计的持久化领域对象就足以提供合适的拓扑信息给仓库管理和物流控制领域对象。类似地，如果整个仓库拓扑可以完全维护在内存中，就不再需要Virtual Proxy和Resource Cache；如果只需要支持一套固定标准的功能集，则一些提高灵活性的机制，如Strategy、Visitor和Interceptor可能也就不需要了。

如果需求和限制条件与我们仓库管理流程控制系统的需求和限制条件相关，那么仓库拓扑设计可以作为其他拓扑管理领域对象的模型。但是，重要的是要仔细检查是否需要支持更多、更少或不同的需求，这样可能就需要采用其他方案。实际上拓扑管理有好几种可行的设计，都是由不同的模式序列来创建，由各自不同的需求集决定的。

模式故事背后的故事

……从此，他们过上了幸福的生活……

这一章，我们抛开仓库管理流程控制系统的细节，再来回顾一下这个例子的全景，看看它是如何指导我们做出设计的。我们将讨论这个例子是如何支持模式语言中的各种属性，以及本书第三部分的分布式计算模式语言是如何支持和指导我们在这个例子选择什么样的模式序列。

回顾一下我们为仓库管理流程控制系统做架构设计的过程，你会发觉对模式的选择和应用是非常自然的，好像本来就应该设计成这个样子，甚至可能会认为是预先设计好的。这个故事自然流畅，我们很容易跟上它的节奏。所以，让人觉得这个架构设计的过程非常直观。

然而，这种感觉实际上是一种过于单纯的错觉，说明你没有意识到存在某种微妙的东西使得设计行为不像转门把手那么简单。对模式序列的分析告诉我们，这是经过深思熟虑的选择，而不是随便拿一个就可以用的。最明显的是，每个单独的模式满足了仓库管理流程控制系统的某个特定需求——这当然是创建一个良好的软件架构的必备条件。然而，仅仅选择了正确的模式还不能保证就能设计出一个高效的、健壮的架构。这些模式必须按照合适的顺序集成在一起，互为补充而不能互相打架。仅仅靠一套各自独立的模式完不成这个任务，因为它们主要关注于解决自己针对的那个问题。

随着设计慢慢地推进，每次引入一个模式，该系统架构的模式序列逐渐创建并显现出来，它不仅是完整的，而且不同的部分互为补充——不仅从功能角度上看是这样，对于成功至关重要的运营和开发方面亦是如此，比如吞吐量、可伸缩性、灵活性和可移植性。

在基线架构以及通信中间件和仓库拓扑的基础结构这个层次，模式序列中的这些模式用于解决系统级别战略性的问题并定义架构主干。像算法上的变化和控制流程这样的局部的战术上的问题则放到了模式序列的后面，等到各自稳定的战略性设计中枢确定下来之后再做处理。我们在序列中每次增加一个模式，增加的时候我们先把它整合到已有的设计中，而不过早地关心起实现细节。换句话说，就是每增加一个新的模式都是将原来的设计进行增强和扩展，从而产生一个新的设计。

所有的这些模式基于其各自的角色集成在一起，它们之间的平衡保证各自能够解决各自的问题，同时又能互相支持从而进一步地表现出各自的能力。例如，我们的基于CCM的ORB中的很多组件分别参与了几个模式的实现。正如我们在第6章通信中间件中所展示的，由Acceptor-

Connector (154) 引入的Acceptor组件, 从Reactor (150) 的角度来看它实际上就是事件处理程序。而从Strategy (266) 的角度看, 它们则是上下文组件, 因为多个Acceptor可以为服务处理程序实现不同的并发策略, 从而使得服务器在侦听新的连接请求的时候可以采取不同的策略。同样, 在第7章中, Storage类实现了如下9个模式中的角色: Composite (185)、Chain of Responsibility (257)、Immutable Value (231)、Proxy (169)、Iterator (173)、Strategy (266)、Visitor (261)、Interceptor (260) 和Resource Cache (299)。这种职责上的合并将很多模式紧密地结合起来, 从而产生了强大而简洁易懂的设计。

其他的模式, 如Layers (108)、Component Configurator (289) 和Strategy (266) 在这个仓库管理流程控制系统的不同部分也担当了设计向导的作用。比如Layers用于在系统基线架构中隔离不同应用功能的不同层次, 同时在基于CCM的ORB的设计中用于隔离和通信相关的不同层次的粒度。在系统的基线层次和通信中间件的内部, 我们都使用了Component Configurator来处理配置和部署功能。Strategy的应用更是遍布整个系统, 包括各种不同的场景和仓库管理流程控制系统的各个层, 我们应用它来支持算法行为和内部中间件机制的变化。

模式可以应用于大规模软件架构中多个层次的抽象和粒度上, 这种能力保证了复杂系统在概念上的完整性。尤其是在整个系统中共同的或者相关的问题可以使用同一个模式解决。于是, 可以说我们所应用的模式序列不仅为仓库管理流程控制系统创建了一个均衡的、可持续的软件架构, 而且它本身也是容易理解和应用的。

此外, 我们所选择的模式序列使得我们可以为仓库系统创建多层的产品线架构, 这种软件架构适用于不同的系统变种和版本。比如, 在领域这边我们支持不同的功能集: 有的系统可能要求整套的仓库管理和物流控制功能, 有些则只需要物流控制功能, 还有一些需要物流控制功能和某些仓库管理功能。相似地, 在基础设施这边, 我们的通信中间件允许系统采用不同的策略进行配置和重配置, 来满足各种不同的应用需求和运营环境。

正是因为我们在选择模式序列进行软件架构的时候遵从了松耦合的原则, 才使得我们的设计具有如此的灵活性。其他和领域相关的变化点包括仓库类型、仓库大小、仓库运营方式。从技术角度讲, 架构——尤其是通信中间件的设计——支持对不同的操作系统和网络协议的集成, 并且可以使用不同的通信和服务质量策略进行配置。该通信中间件的设计和实现独立于仓库管理流程控制领域, 所以这也是一份可重用的资源, 它可以用在很多其他领域的分布式系统中。

由此我们可以得出结论, 仓库管理流程控制系统的架构的关键质量因素来自一个成功的、可靠的模式序列。这个模式序列可以帮助我们采用成体系的方式应对问题和构建系统。然而, 它只是我们的分布式计算模式语言中的可行的模式序列之一, 即使对于仓库管理流程控制系统也不只这样一种设计。我们可以看到对于一个模式语言而言, 要能够更好地解决某个领域的问题, 它必须支持很多很多的模式序列。也就是说, 模式语言的质量是由其中能够达到的序列的质量决定的。也只有这样的模式语言才能够帮助我们找到某个特定领域中开发软件的主要挑战, 并且告诉我们按照什么样的顺序去应对这些挑战, 在应对挑战的时候存在哪些备选的设计, 以及何时选择哪种设计去解决这些挑战。

模式语言将这些优秀的模式序列组织成协调一致又互为补充的集合, 因此它便成为我们设

计、实现、自定义和应用可重用的产品线架构、平台架构和框架的重要工具。反过来，模式故事又帮助开发人员理解已有的软件的设计和实现，以便他们可以在自己的项目中高效地使用这些系统（比如使用现成的通信中间件），或者维护和改进软件（比如我们的仓库管理流程控制系统）以满足新的需求。

Part 3

第三部分

模式语言

“我希望生命不是这样短暂”，他想，“语言是如此耗费时间，人们想了解的所有事物均是如此。”

——J.R.R. Tolkien, 英国语言学家、作家, 《失落之路》

在本书的第三部分, 我们提供了一个可行的分布式计算模式语言。它既是我们自己实现分布式系统时的经验总结, 同时也参考了那些经验丰富的架构师、设计人员和开发人员向社区贡献的分布式模式, 我们对其进行了提炼。模式语言可以用于开发很多现实中的分布式对象计算中间件以及分布式应用。你可以和同事以及项目合作伙伴用它来指导设计新的分布式系统, 也可以用来改进和重构现有系统。

在过去的15年, 我们参加了许多产业的网络化、并发性以及分布式系统的开发工作, 从工业流程自动化系统、医疗成像和大型电信系统, 到高性能通信中间件等。我们在书中这部分给出的分布式计算模式语言将这些经验提炼成有实际意义并能直接使用形式。你可以用它来构造新的分布式系统, 改进、重新规划或重构现有系统, 或者用于理解工作中所使用分布式软件系统和中间件的架构。

我们的分布式计算模式语言共包含114个模式, 这些模式被分成13个问题域。每个问题域解决构建分布式系统中的某一个特定技术主题, 并且模式语言中的所有模式组合起来应对与技术主题有关的挑战。引入问题域的主要目的是为了更好地了解模式语言和相关模式, 针对相关问题的模式都放在公共且范围清楚的上下文中讨论。这些问题域按照它们在构建分布式系统时的相关性的大致顺序进行描述。

每个问题域及其组成模式构成单独的一章。

- 第9章从混沌到结构, 包含分布式计算模式语言中的10个根模式。它们帮助我们从项目开始时需求和限制的混沌中逐步提炼出粗粒度的软件结构, 将待开发系统划分成单独的、可处理的部分。
- 第10章分布式基础设施, 描述了和中间件、分布式基础设施相关的12个模式, 从而有助于简化分布式应用。

- 第11章事件分离和分发，包含4个模式，提供了用于灵活有效地分离、转发以及响应网络上接收事件的基础设施。
- 第12章接口划分，提供了11个模式帮助我们设计和指定有意义的组件接口，从而既方便常用的组件使用场景，还能支持特殊目的和额外的场景。
- 第13章组件划分，包含6个划分组件的模式。这些模式的焦点是支持可见的组件质量属性，如性能、可伸缩性和灵活性。
- 第14章应用控制，描述了8个模式，用于将用户输入转换成对应用的具体功能服务请求，执行这些请求并将结果转换成对用户有意义的输出——这称得上是一项很有挑战性的任务。
- 第15章并发，由4个关于并发性的模式组成，帮助服务器和服务器端软件处理多个客户端同时发出请求的情况。
- 第16章同步，描述了9个关于同步访问共享组件、对象和资源的模式，通过总结高效的同步策略或最小化同步的需要来实现。
- 第17章对象之间的互操作，由支持应用中交互式组件和对象间有效协作和数据交换的8个模式组成。
- 第18章适配与扩展，描述了13个模式，以便长期存在的系统——特别是分布式系统——中的组件和对象能适应各自的配置、调整和改进。
- 第19章模态行为，提供了3个模式，以解决本质上是状态驱动的组件和对象的结构化问题。
- 第20章资源管理，包含在分布式系统中用于组件和资源显式管理的21个模式。
- 第21章数据库访问，以本章5个模式作为我们模式语言的“结束”，提供从面向对象应用设计到关系型数据库的有效映射，而不需要引入两者之间的紧密依赖。

我们在本书中讲述分布式计算模式语言的主要目的介绍或引出当前构造分布式系统主要领域的最佳实践和最新技术。然而，它并不是有关分布式计算的全面教程，而是专注于分布式软件系统的设计。因此我们假定读者对分布式计算的核心概念和机制有一定程度的了解，这些概念和机制可以从相关文献[TaSte02][Bir05]中获得。



位于阿特拉斯山脉上的埃本哈杜古城, 联合国教科文组织世界文化遗产
©鲁茨·布施曼

本章给出了分布式计算模式语言的根模式和切入点。这些模式有助于我们从最初的需求和约束条件的混沌中提取出粗粒度的软件结构, 将其划分为构成待开发系统的有实际意义的各个部分。本章同时也描述了设计出可持续软件架构的几个关键因素: 运行方面因素(如性能和可用性)和开发质量因素(如扩展性和可维护性)。

大的分布式系统很容易变得复杂。刚开始, 我们所拥有的只是关于要开发的软件系统的一系列需求和约束条件。一个不成熟的开发方案, 其结果很可能是“一个大泥球”[FoYo99]、一个设计和编码都很混乱的软件, 这会导致无法分辨其内在的架构。此类软件难于理解、维护和改进, 并且随着时间的流逝很容易在稳定性、性能、可伸缩性, 以及其他核心运行架构质量[Bus03]上陷入困境。

决定软件开发成功与否的一个重要因素是结构。结构应该易于被开发人员理解, 应该对系统及其开发所面临的压力具有弹性, 应该支持围绕它而推进的开发过程, 应该兼顾使用并维护它的

组织和个体的需求。简单来说,这种结构能给软件系统的开发人员和其他涉众提供一个良好的环境。然而,如果软件的结构设计没有远见和适当的指导,它将很容易变得混乱,而且还会导致不仅系统的全局难以把握,甚至局部也是如此——代码中充满零碎的细节和假设而难以理解。

因此,在这样的软件开发过程中,我们需要一个粗粒度的系统概念——在抽象和分离的帮助下——可以忽略不必要的细节,在更广的层次上组织系统的核心概念。

首先,软件架构必须是对系统应用领域有意义的表现。特别是系统提供的功能和特性必须支持具体的业务,否则对用户来说就没有实际价值。然而,如果系统的软件架构没有适当描述其应用领域及范畴,那么它也很难(如果不是不可能的话)提供用户级服务和特性以满足系统功能需求。

为系统功能架构建模需要更进一步考虑的问题是可变性。变化可能是因为不同的特性集、业务流程的改变、具体业务算法的选择,以及系统用户界面的不同选项。如果不能明确地知道应用领域中哪些东西会发生变化,或者必须支持什么样的变化,就很难为软件系统或产品提供适当层次和程度的灵活性。

我们模式语言的根模式阐述了如何应对挑战以构建应用Domain Model,从而不仅反映软件系统的功能需求,还能为技术架构的进一步细化打下坚实的基础。

Domain Model模式(106) [Fow03a]定义了一个应用领域结构和工作流的精确模型——包括它们的变化。模型元素是应用领域有意义的抽象,角色和交互反映了领域工作流和对系统需求的映射。

图9-1说明了Domain Model是怎样和我们的分布式计算模式语言主体连接起来,以及如何协调这些模式来细化自身的。

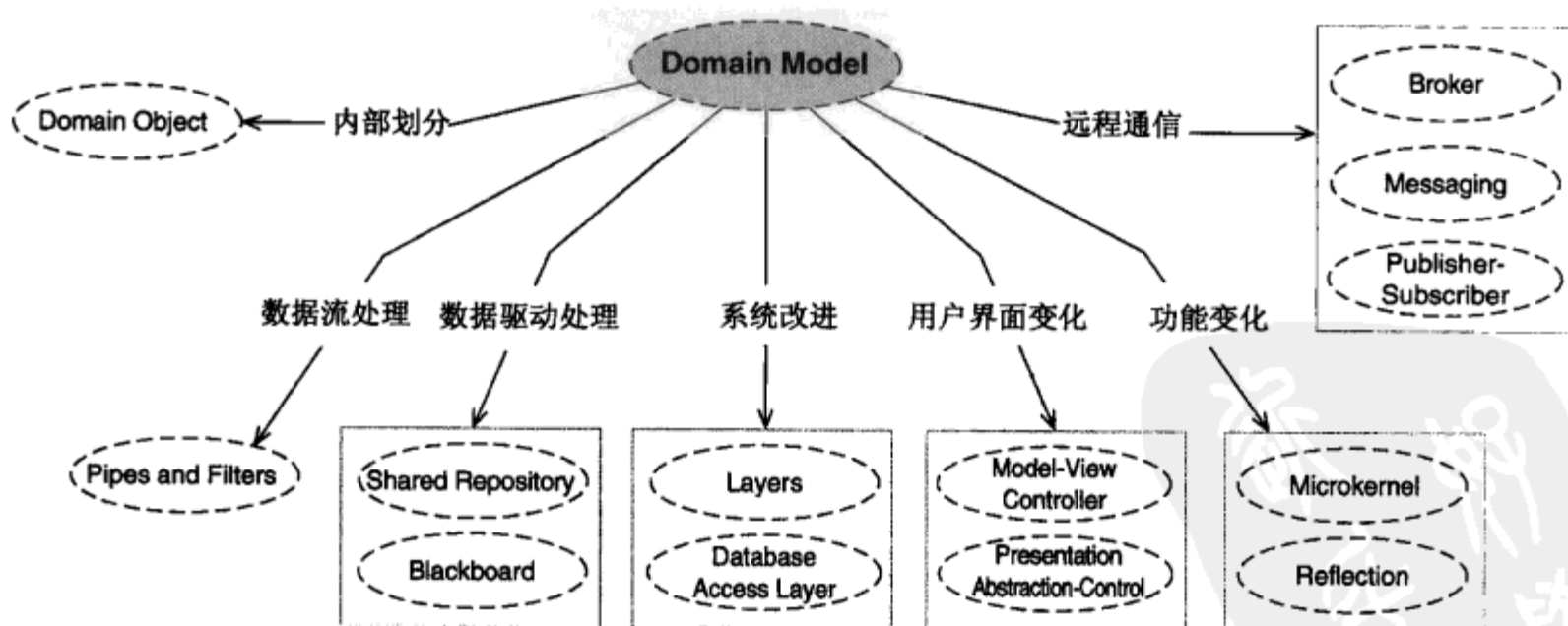


图 9-1

Patterns of Enterprise Application Architecture [FOW03a] 中也描述了 Domain Object, 但是那里仅关注于 Domain Model 实现的技术方面, 而没有明确描述领域建模问题。

然而, 仅仅将分布式软件系统的核心结构从应用领域可见角度进行划分并不能保证定义出一

个可行的基线架构。一方面，软件系统需要包括许多组件，并表现出许多与领域无关的属性。例如，服务质量需求（如性能和可预测的资源使用）是一些交叉性的问题，因此不能只通过组件划分来解决。类似地，及时响应用户交互的需要也可能与网络延迟和部分失败模式相冲突。另一方面，我们作为开发人员希望构造的系统不是简单地满足用户可见需求就可以了。用户对开发质量，如可移植性、可维护性、可理解性、扩展性、可测性等并不关心，但开发人员却不能这样。

找到一个合适的应用划分方式与下面几个带有挑战性的关键问题的答案有关。

- 应用怎样和其环境交互？某些系统与不同类型的用户交互，另一些和其他系统交互，还有其他一些系统则嵌入在更复杂的系统中。不可避免地还有一些系统具有所有上述交互方式。
- 应用处理是怎样组织的？某些应用从客户端接收请求并作出反应和响应。另一些应用处理数据流。某些应用完成自定义的任务而不需要外界激励。事实上，对于某些应用来说，我们甚至无法识别出具体的工作流及其组件间的显式协作。
- 应用必须支持什么样的变化？灵活性是软件开发中需要考虑的主要问题之一，特别是当开发软件产品或产品系列时，这时需要满足不同客户端的广泛需求。某些系统必须支持不同的特性集，比如提供给小型、中等和大型企业，以满足不同的市场和客户群。另一些系统必须支持多种业务流程，这样每个客户可以根据自身特殊业务来合理定制 workflow 模型。还有一些系统必须支持算法行为和可视界面的多样性，以吸引更多广泛的客户群。
- 应用的预期寿命是多少？某些系统是短期应用并会在不需要的时候扔掉，如探索短期市场趋势的在线交易程序。另一些系统可能要运行30年或更长，而且必须对变化的需求、环境和配置作出响应，如电信管理网（TMN）系统。

因此，我们的分布式计算模式语言包含了9个全局性的模式，以帮助将Domain Model（领域模型）转化成技术软件架构，从而作为进一步开发的基础。每个模式为上述问题给出了自己的答案。

- Layers（层）模式 (108) [POSA1]帮助我们将应用结构分解为子任务群，每个子任务群是按照特定的抽象层次、粒度、硬件距离（hardware-distance）或其他标准来划分的。
- Model-View-Controller（模型—视图—控制器）模式（MVC）(109) [POSA1][FOW03a]将交互式应用分成3个部分。Model部分包含核心功能和数据；View部分为用户展示信息；Controller部分处理用户输入。View和Controller一起构成用户界面。采用变化传播机制来确保用户界面和模型之间的一致性。
- Presentation-Abstraction-Control（表现—抽象—控制）（PAC）(111) [POSA1]按照合作代理（cooperating agent）的层次结构来定义交互式软件系统的结构。每个代理负责应用功能的特定方面，并包含3个组件：Presentation、Abstraction和Control。这种划分将代理的人机交互部分与功能核心以及与其他代理的通信分离开。
- Microkernel（微内核）模式 (113) [POSA1]适用于软件系统必须适应不断变化的系统需求的场合。它将最小的功能核心与其他扩展功能以及用户定制部分分离开。Microkernel同时提供集成扩展插件并协调彼此协作的功能。

- Refelection（反射）模式 (114) [POSA1]提供了一种动态改变软件系统结构和行为的机制。它支持功能方面的改变，如类型结构和函数调用机制。该模式将应用分为两部分。基础部分包含核心应用逻辑。其运行时行为可以通过维护所选系统属性的元数据来识别，从而使得软件系统具有自我识别能力。因此，改变保存在元数据中的信息可以改变接下来的基础行为。
- Pipes and Filters（管道和过滤器）模式 (116) [POSA1][HoWo03]提供了可以处理数据流的系统结构。每个处理步骤都封装在一个Filter组件中。Pipe用于将数据在相邻的Filter中传递。
- Shared Repository（共享仓库）模式 (117) [HoWo03]帮助我们结构化那些功能和协作均由数据驱动的应用。Shared Repository负责维护应用组件操作的公共数据，这些组件在Shared Repository中能够自行访问并修改数据，此外，Shared Repository中的数据状态影响特定组件的控制流。
- Blackboard（黑板）模式 (119) [POSA1]对于那些没有已知的确定解决方案的问题非常有用。在Blackboard实现中，几个专门的子系统汇集起来，提供一个可能的局部解或近似解。
- Domain Object（领域对象）模式 (121)将自我完备的连贯功能和基础性责任封装成定义良好的实体，通过一个或多个Explicit Interface提供功能，并隐藏内部结构和实现。

我们在将Domain Model提炼成可持续的软件基线架构时，上述9个模式阐述了针对各种不同问题的对策。

Layers定义了一个根据（子）系统级属性划分应用职责的通用办法，例如按照每个功能组进行独立地开发和改进。我们可以定义一维或多维的划分标准，如抽象、粒度、硬件距离以及变化率。Layers是在软件架构中划分和组织不同关注点的最基本的模式之一。

Layers和我们分布式计算模式语言集成在一起，如图9-2所示。

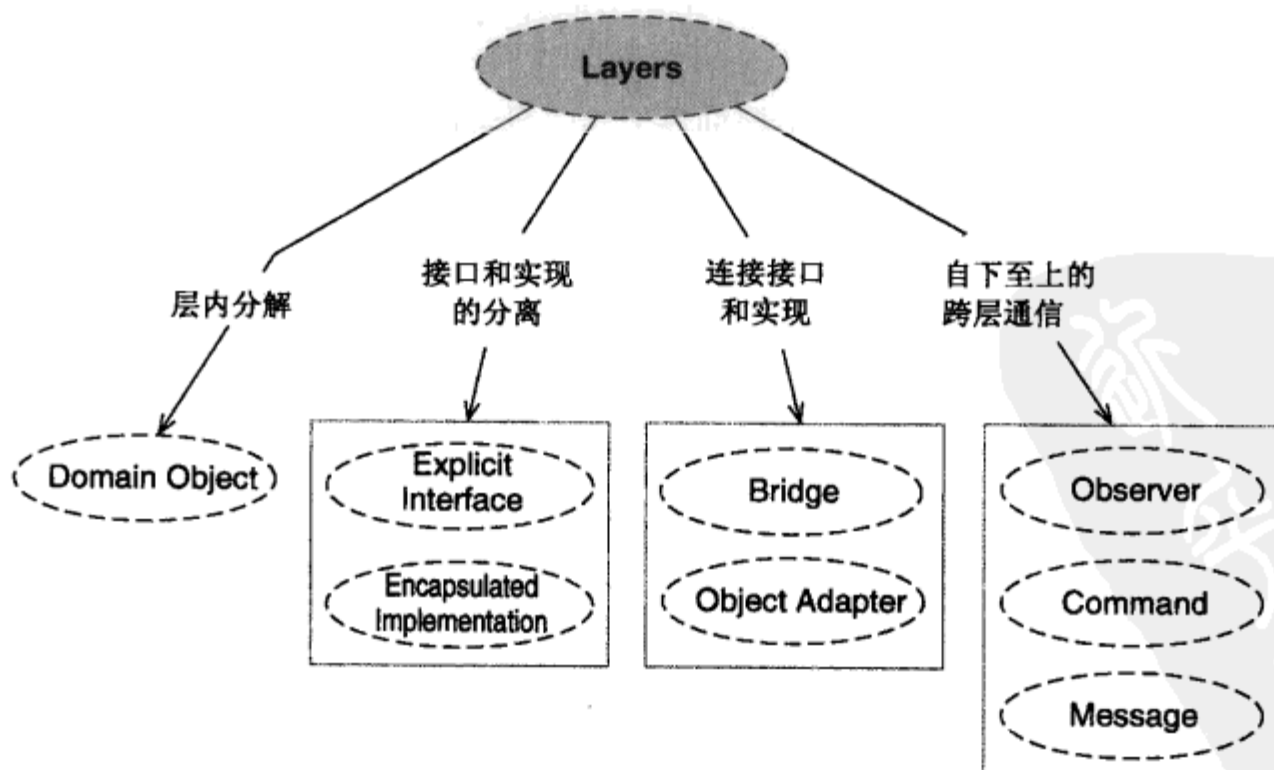


图 9-2

接下来的两个模式，Model-View-Controller和Presentation-Abstraction-Control，主要解决对用户界面变化的支持问题。尽管这两个模式在很多方面都是相关的，它们并不总是可以互相替代的。简单来说，Model-View-Controller支持某一特定用户界面的变化，而Presentation-Abstraction-Control支持使用多个不同的用户界面，并允许它们各自独立地变化。因为绝大多数软件系统只需要一套用户界面风格，Model-View-Controller应当是首选方案。

与此相反，当软件系统被分为多个、大部分时间互相独立偶尔有交互的子系统，并且每个子系统都需要自己的用户界面风格时，Presentation-Abstraction-Control才比较有用。这样的例子包括在船上和水下都要使用的离岸软件、机器人控制系统或部分通过虚拟现实设备操作的应用等。只有很少的系统属于这一类，因此应用Presentation-Abstraction-Control的场合要比使用Model-View-Controller的场合少得多。

图9-3阐述了如何将这两个模式集成到我们的分布式计算模式语言中。

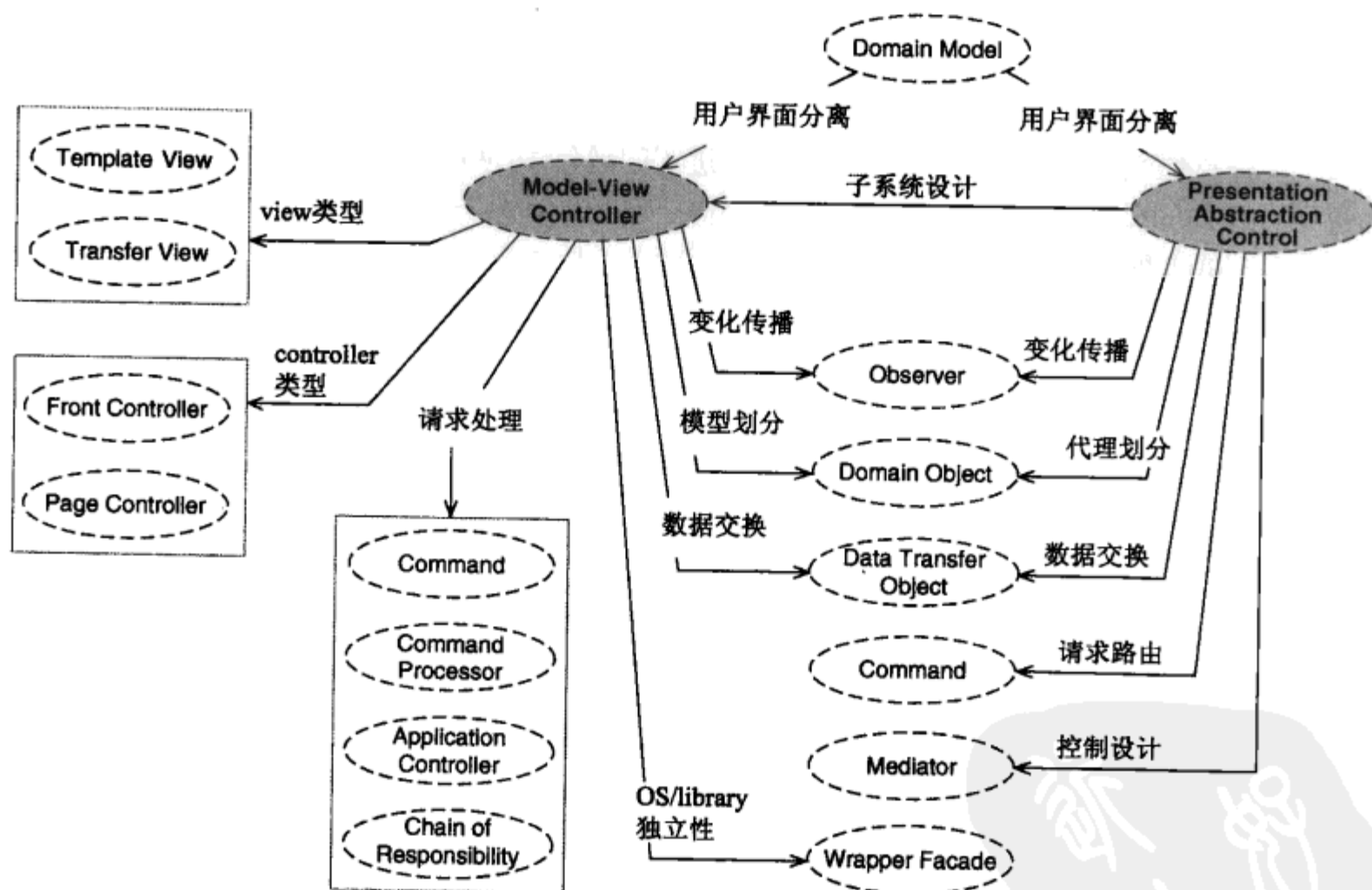


图 9-3

Model-View-Controller模式在*Patterns of Enterprise Application Architecture* [Fow03a]中也有描述，其目标结构和行为同POSA中的版本是一样的。Model-View-Presenter模式[FOW06]也和Model-View-Controller相关，但它更适合在富客户端开发中使用，因为它并不把所有View的行为都交给Model来完成，比如当用户点击“应用”按钮时，它能自己判断是否能不通过Model而直

接处理。这一点增强了复杂View的可组合性以及不同角色的可测试性。

接下来的两个模式，Microkernel和Reflection，都关注于软件系统的灵活性。然而，这两个模式关注的角度又有所不同。简单地说，Microkernel提供基于插件的架构，它所支持的灵活性在于系统为用户提供什么功能。因此Microkernel逐渐发展为操作系统、中间件以及产品线架构等方面的流行架构。

不同的是，Reflection定义了一种架构，将系统的结构和行为的特定方面对象化，它所提供的灵活性在于系统功能如何执行以及客户端怎样使用这些功能。因此，它通常在应用环境和服务集成的场合中使用，在这些情况下客户端应用必须能使用和控制其他应用的功能，同时又不需要明确地知道其内部接口和行为。

图9-4和图9-5展示了Microkernel和Reflection是怎样与我们的模式语言中的其他模式联系在一起的。

Pipes and Filters模式适用于处理数据流的应用，或者组件通过交换数据流来通信的场合。图像处理是一个典型的例子，其Domain Model能很好的由Pipes and Filters架构表现。

Pipes and Filters模式与我们模式语言的集成如图9-6所示。

有些资料把Pipes and Filters模式也看作是进程间通信[HoWo03][VKZ04]的一种基础方式，但在我们的分布式计算模式语言中，我们更倾向于它是应用服务之间组织协作的一种机制，而非交换信息的一种方式。在Pipes and Filters的实现中，后者通过消息来完成。正如我们在第10章中指出的，消息同远程方法调用和发布者/订阅者一样重要，它们共同构成三种最主要的通信方式。

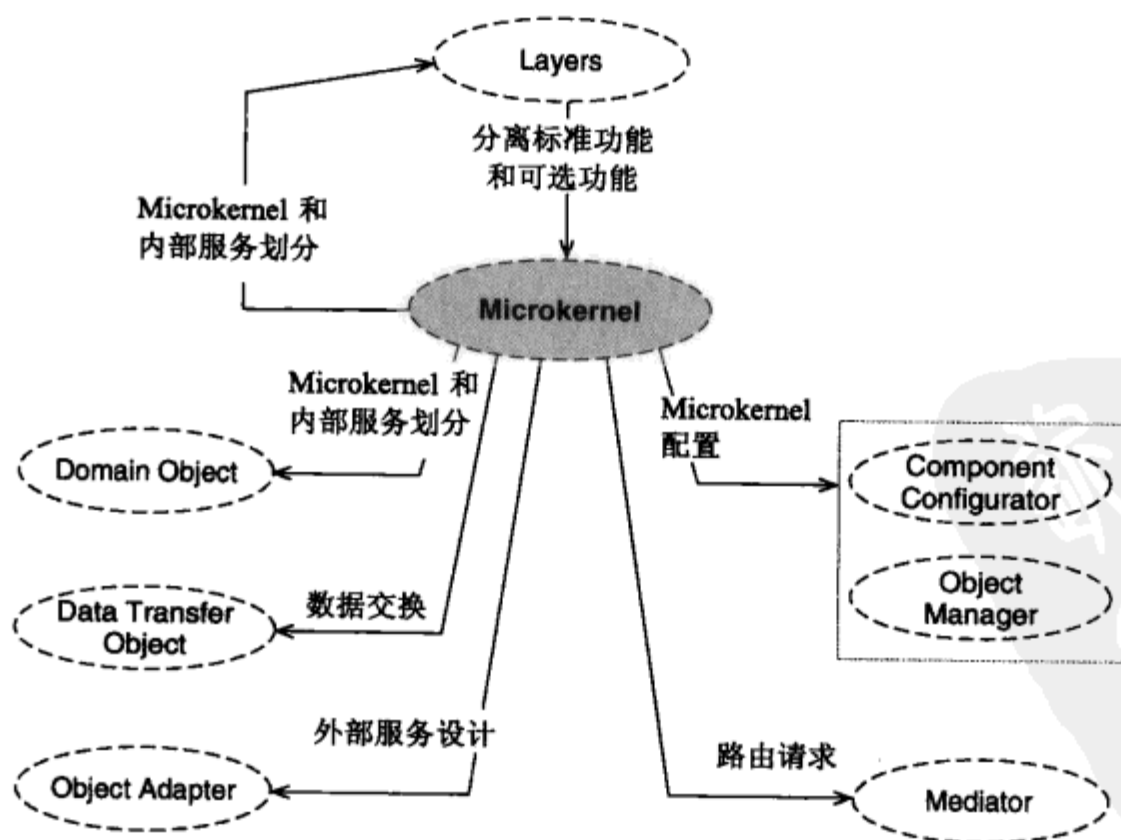


图 9-4

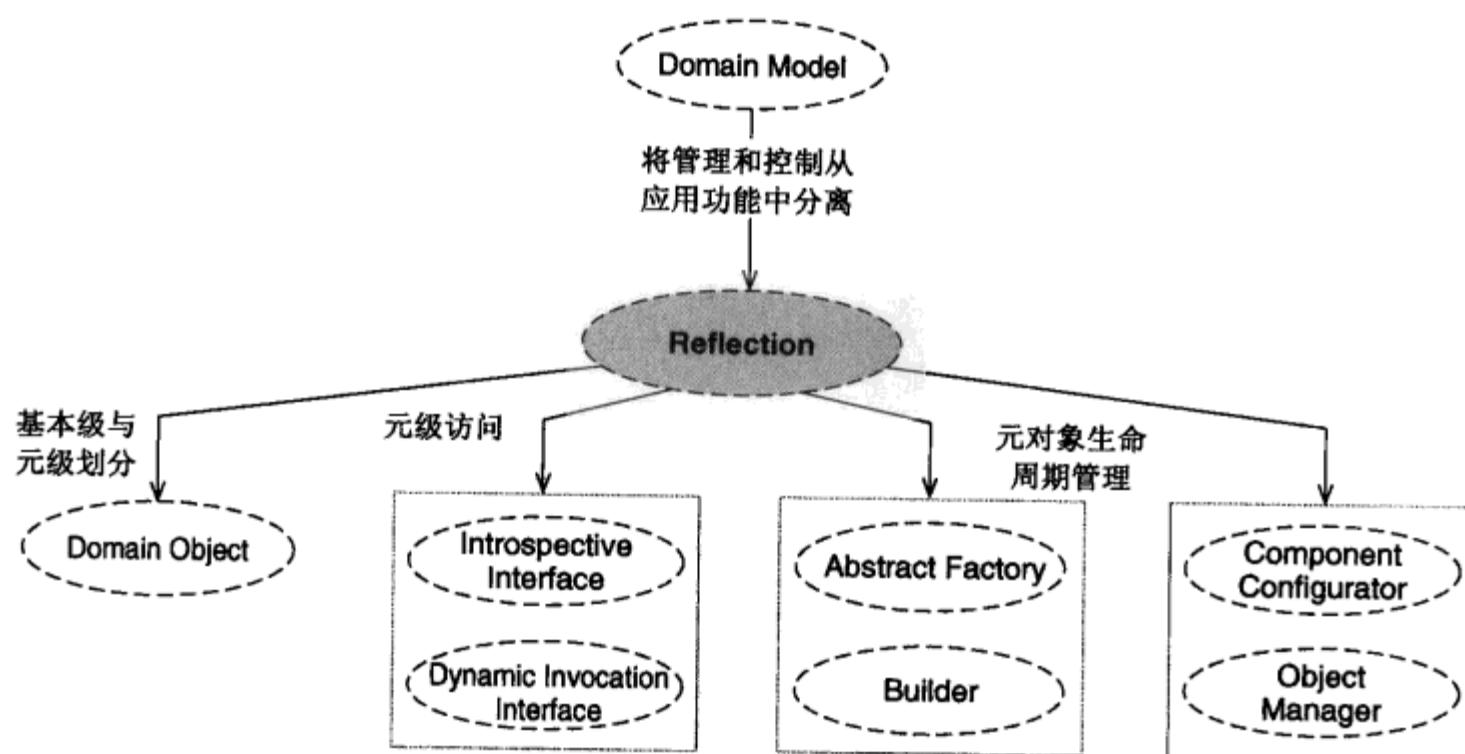


图 9-5

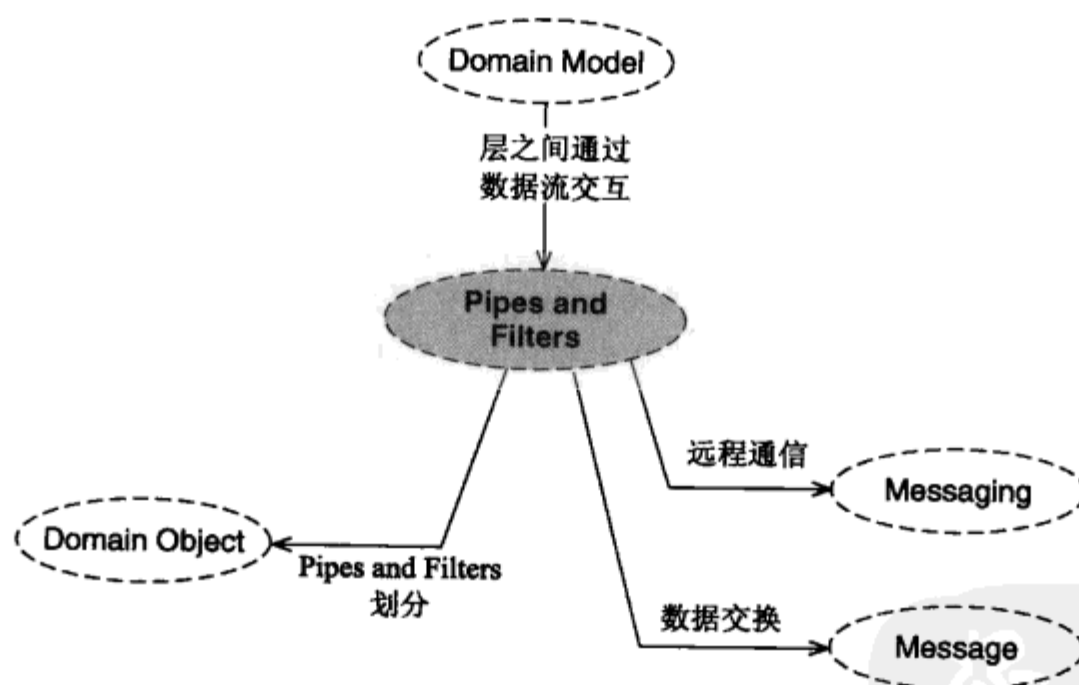


图 9-6

Enterprise Integration Patterns [HoWo03] 中介绍了另一个版本的 Pipes and Filters，那里它用来在面向消息的中间件中分步转换消息格式和内容。POSA 中的 Pipes and Filters 的含义更广，因为它用来定义整个数据流处理应用的结构。我们的模式语言中对 Pipes and Filters 的使用符合 POSA 中的范畴。

有些应用它们的组件主要处理一套公共的（结构化）数据，我们可以在设计中使用 Shared Repository 和 Blackboard 模式。通过分离系统中的数据与功能，应用组件间的数据交换可以简化，并且可以通过数据值的改变（Change of Value, CoV）通知来实现组件间相互协调。

这两个模式和我们分布式计算模式语言的集成如图9-7所示。

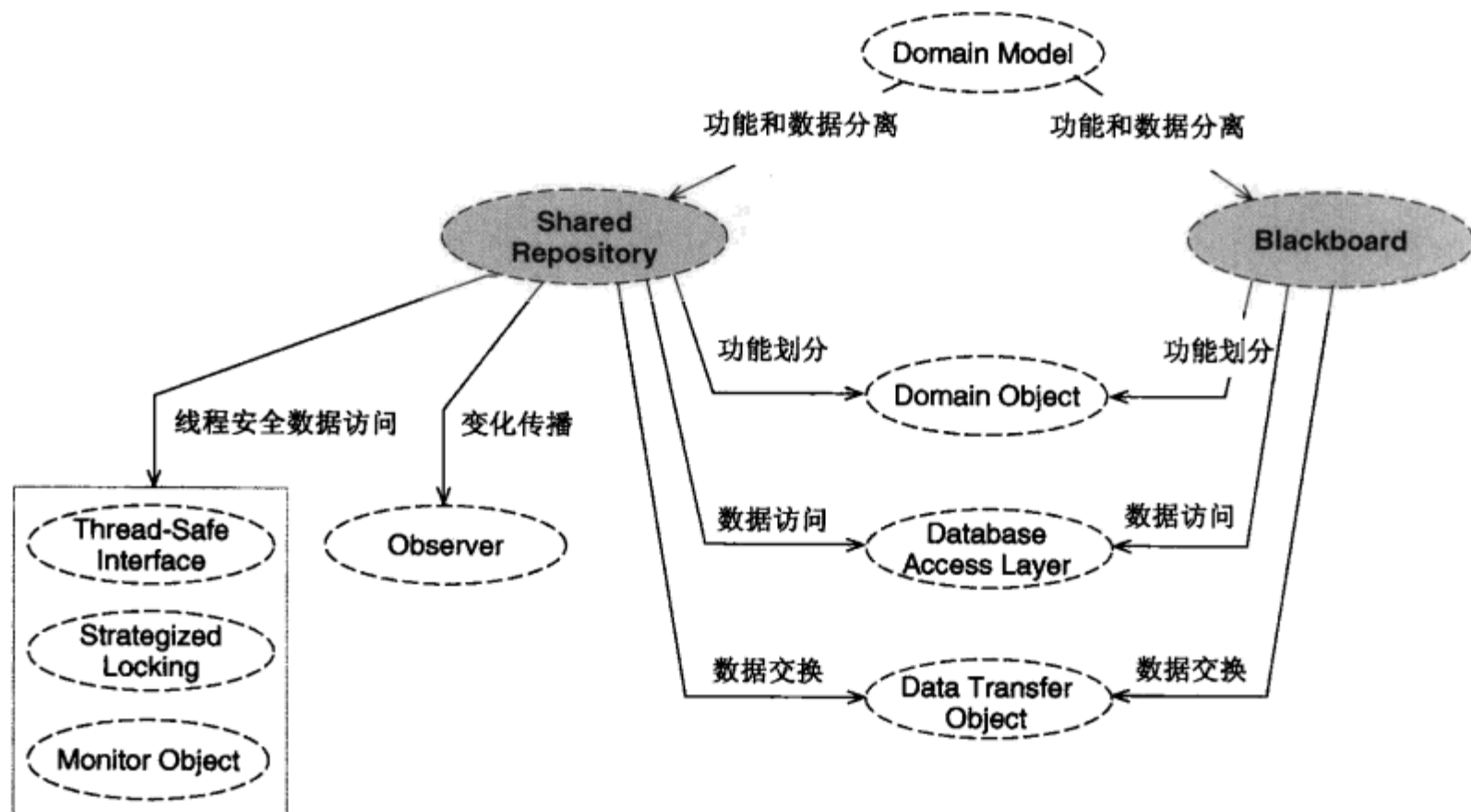


图 9-7

这两个模式的不同在于它们的计算机制。在 Shared Repository 架构中，应用组件实现确定的控制流并按明确编码和配置的方式合作。因此它主要适用于网络管理和控制系统领域的应用，这时往往需要操作外围设备产生的大量数据，比如电信管理网（TMN）系统或工业流程控制系统。

Blackboard 架构则与之不同，它实现了基于启发式方法的计算模型，可以在应用领域不存在已知的确定可行的算法时，或者当输入数据混乱、不准确或者质量可疑的情况下产生有用的结果。例如，Blackboard 在生物信息系统中相当流行，而生物信息系统通常都在大量杂乱、不完整或部分错误数据的基础上运行。它还曾经在语音识别应用上很流行，直到发现了更合适的确定性解决方案。

一般来说，Blackboard 是 Shared Repository 的一种特殊变体，但是它有自己的一套不同的驱动因素，所以其解决方案也就需要不同的运算机制。尽管 Blackboard 的应用范围比 Shared Repository 要窄，但这些不同仍使其可以作为一个单独的模式来描述。

Enterprise Integration Patterns [HoWo03] 中也有关于 Shared Repository 模式的描述。在该书中它主要关注（企业）应用集成，而不是协调数据驱动（Data-Driven）应用的控制流，而后者正是我们的模式语言所关注的范畴。与 Pipes and Filters 一样，某些资料也把 Shared Repository 作为进程间通信 [Fow03a][HoWo03][VKZ04] 的基本方式，在分布式计算模式语言中我们不这么认为，而是把该模式作为划分应用功能的一种机制。

本章中我们给出的最后一个模式Domain Object支持将应用中自我完备的职责封装在一个明确的软件实现中。通过这种封装，我们可以明确、直接地满足这些职责的功能、操作和开发方面的需求，并独立于其他Domain Object的实现。

图9-8描绘了Domain Object和我们分布式计算模式语言的集成。

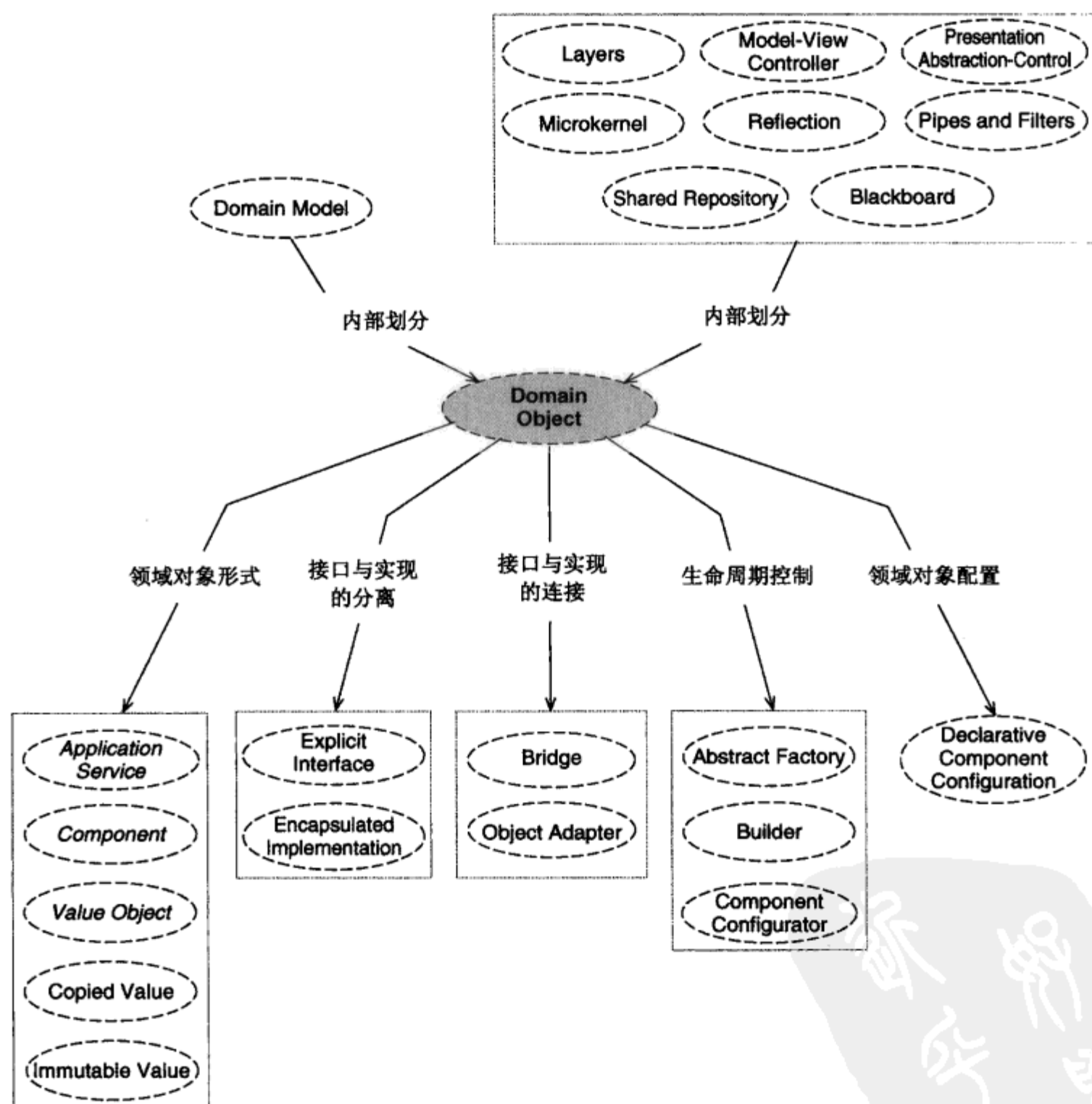


图 9-8

有些引用到的模式没有在本书中进行描述，但却属于我们的模式语言所包含的范畴，因为它们代表了不同的Domain Object形式。Application Service是Core J2EE Patterns中的关键模式之一，

用来划分企业应用的业务逻辑[ACM01], Component是*Server Component Patterns*中的两个根模式之一[VSU02]。因此, 这两个模式将我们模式语言与*Server Component Patterns*和*Core J2EE Patterns*中包含的所有模式联系起来。最后, Value Object是一种由状态而不是类型来唯一标示自身的小对象[PPR][FoWo03a]。

现实世界中的绝大部分软件系统都不是由上述应用划分模式中的单一模式来构成的。不同的模式提供不同的架构属性, 而一个系统很可能需要由多个全局性的模式发展而来, 以满足其系统需求。例如, 可能你需要构造这样一个系统, 它有两个不同的设计目标并偶尔会互相冲突, 如用户界面的可适应性以及多个平台的可移植性。对于这样的系统, 你必须将几个模式结合起来才能形成一个合适的结构。当扩展到分布式环境时, 这些应用基础设施模式也必须与适当的分布式模式集成起来。

选择应用划分模式或组合几个模式, 只是设计软件系统众多步骤中的一个。选择划分模式并不意味着完成了系统软件架构, 或解决了影响设计风格形成的所有重大决定。相反, 这只是软件系统的一个结构框架, 必须进一步细化和改进。这个过程包括在框架内展现应用的核心功能, 细化组件及其关系。我们会在模式语言后续章节中的模式来阐述这些过程。

9.1 Domain Model**

在开始构建(分布式)应用的时候……我们需要确定待开发软件的初始结构。



需求和约束条件告诉我们软件系统的功能、服务质量以及部署方式, 但是并不能为我们提供一个具体的结构来指导开发。在对系统的范畴和应用领域进行严谨而深入的分析之前, 系统的实现在很大程度上像“一个大泥球”, 很难理解或传达给客户, 这只能为要构造的系统提供一个很差的架构基础。

需求列表给出了应用的问题领域, 而不会给出其解决方案。对于一个运行中的系统, 其需求必须由具体的软件实体来实现。如果这些实体及其交互与应用的核心业务无关, 那么它将难以理解系统的真正功能, 自然也难以表现出系统真正的功能。相似地, 这将会导致系统难以满足服务质量需求, 因为这些需求很难清楚地映射到相关的软件单元。因此, 若对系统的应用领域没有一个清晰的视图, 软件架构师将无法确定他们的设计是否正确、完善、内聚, 也就无法确定将设计作为开发基础是否足够合适。

因此, 创建一个模型来定义系统的业务职责及其变化范畴, 模型元素是对应用领域的有用抽象, 其角色和交互反映了该领域的工作流(见图9-9)。

软件架构在自身不断改进的过程中成为模型的表现, 而Domain Model则是其基础。

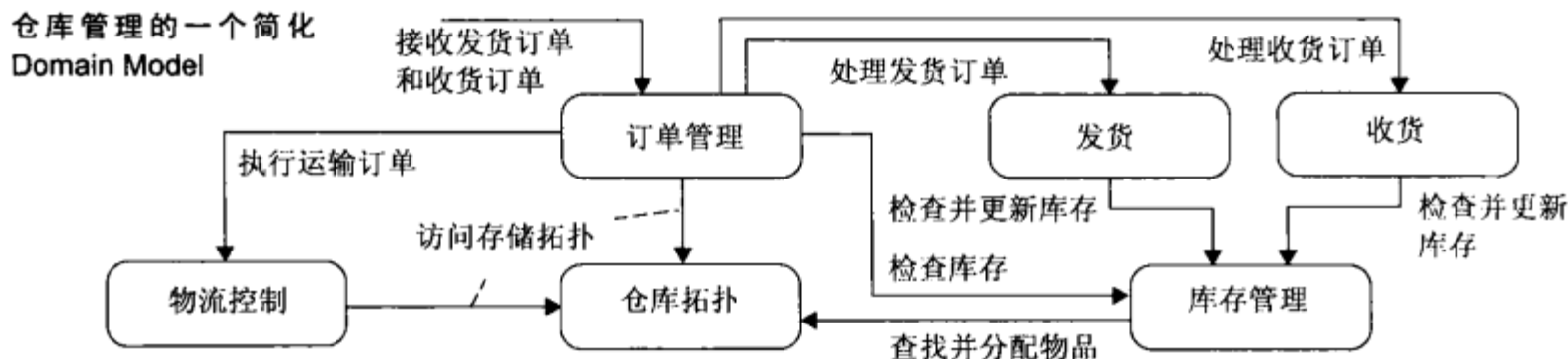


图 9-9



Domain Model迈出了将需求转换成可持续的软件架构和实现的第一步。它为应用领域的结构和工作流定义一个精确的包括相应变化的模型，这有助于将需求映射为具体的软件实体并检查需求是否完善和一致。我们可以发现遗漏的需求、澄清混乱的需求以及去除不必要的需求。这样就能清楚地设定系统架构的职责和界限。一个好的Domain Model还能更容易满足系统的服务质量需求，因为这些需求能够被分配给模型中所应用的特定元素和工作流。Domain Model还有助于促进软件专业人士、领域专家以及顾客之间的沟通，因为其组成元素是建立在应用领域的术语之上的。

一般来说，我们可以通过适当的方法来创建Domain Model，比如领域驱动设计[Evans03]和领域分析[CLF93]。还有一些专门表现领域变化的方法，如普遍性/多样性分析[Cope98]和特性建模[CzEi02]等。领域相关的模式能进一步支持Domain Model的创建，它们可以提供该领域中公共抽象和工作流所重复采用的经典解决方案，包括这些解决方案可能的相应变体。很多领域都记录了领域相关的模式，如电信、医疗保健以及公司财务等[Fow97][Ris01][PLoPD1][PLoPD2][PLoPD3][PLoPD4][PLoPD5]。

Domain Model达到一定的成熟度时，能充分展现应用的功能职责及其相应的变化，那么下一步就可以将模型转换成具体的架构，来表现和支持这一功能，并解决一系列的服务质量需求，如性能、可伸缩性、可用性、适应性和扩展性。

有些模式有助于合理安排并连接Domain Model中的各元素以支持特定形式的计算。例如，Pipes and Filters (116) 适合处理数据流的应用；Shared Repository (117) 能帮助我们组织数据驱动的应用；而Blackboard (119) 则比较适合那些运行在不完整或混乱的数据上的应用，或者是没有已知或可行的确定解决方案和算法的情况。

另外一些模式能帮助我们z将Domain Model中的元素组织在一起或分离开，以增强系统在特定方面的适应性、扩展性以及未来的改进。例如，Layers (108) 将Domain Model中具有相似职责、属性或粒度的元素划分到不同的层中，这样每一层都可以独立改进。Model-View-Controller (109) 和Presentation-Abstraction-Control (111) 将用户界面和领域功能分开，这样就可以在不修改业务逻辑实现的情况下适应用户相关的界面变化。Microkernel (113) 将应用划分为核心功能、版本相关的功能和版本相关的API，以支持产品的差异。Reflection (114) 将系统结构和行为的特定方面展现出来，从而可以在运行时改变功能执行并且能被客户端所使用。最后Database Access Layer (318) 将应用功能与关系数据库分离开，从而方便数据库的更换。

在产品系统中，上述的几种模式可以结合从而形成应用的基线架构。例如，Model-View-Controller也许能和Reflection以及基于Shared Repository的计算模型结合在一起。

通常，应用基线架构中每个自我完备且内聚的实体或职责都由一个单独的Domain Object来表示，以提供固定的软件实现来满足其在功能、运行以及开发上的特定需求。

在分布式系统中，应用基线架构中的Domain Object能通过中间件通信。例如，Broker (137) 支持应用组件通过远程方法调用来通信，Messaging (129) 支持在系统组件间交换异步消息，而Publisher-Subscriber (135) 通过状态变化通知在组件间通信并协调彼此的处理过程。

9.2 Layers**

在将Domain Model (106) 转化成软件的技术架构的时候，或者在实现Broker (137)、Database Access Layer (318)、Microkernel (113) 以及Half-Sync/Half-Async (209) 的时候……我们必须支持系统不同部件之间的独立开发和升级。



不管软件系统不同部分之间有什么样的交互和耦合，我们都希望能对其进行独立的开发和改进，这可能跟系统的大小有关或者是上市时间方面的要求。然而，如果对系统软件架构的不同方面没有一个清晰合理的划分，就很难支持各部分之间的合理交互，进而也无法独立地开发。

我们所面临的挑战是必须找到一种平衡，使得我们的设计既能够将应用划分为有意义、可处理的部件，使它们可以独立开发和部署，同时又不能将自己淹没在太多的细节中，从而丧失对整个架构远景的把握或者带来运行问题，如性能、可伸缩性等。专用的整体性设计已经无法应对这个挑战。尽管它能更直接地解决服务质量方面的问题，却很容易产生意大利面条式的结构，从而降低开发质量，如可理解性和可维护性。

因此，要为待开发系统定义多个Layers，每一层都有明确而具体的职责（见图9-10）。

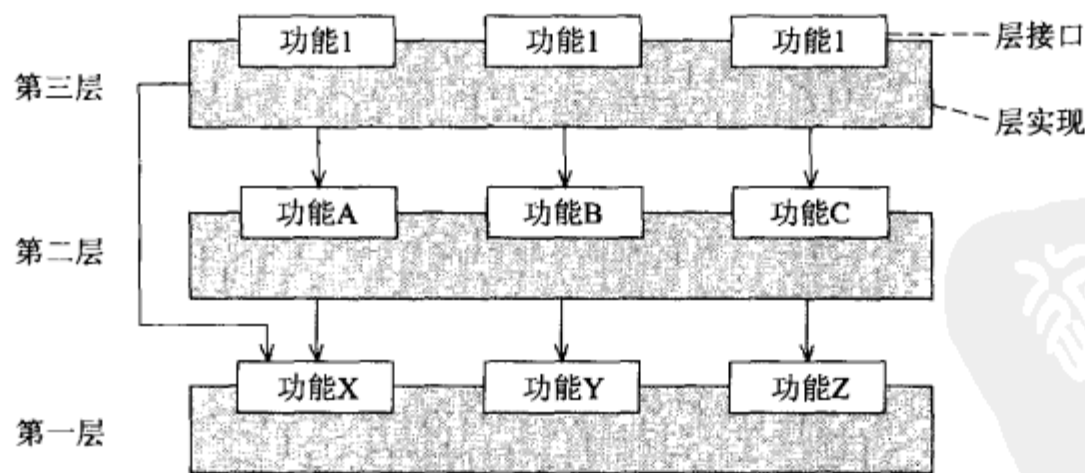


图 9-10

将系统功能分配给各个层，并且让每个层的功能只建立在同层或较低层所提供的功能之上。为所有层提供接口并与它们的实现隔离开，在每一层中程序只通过这些接口来访问其他层。



Layers架构根据（子）系统级属性定义了软件功能的横向划分，从而将每组功能都清晰地封装起来以便独立改进。具体的划分标准可以通过各种不同的维度来定义，如抽象性、粒度、硬件距离以及变化率。例如，将架构划分为表现层、应用逻辑层和持久化数据层的分层方法是从抽象性维度来划分的；引入业务对象层，其内部实体单元供业务处理层使用，这种分层机制是按粒度来划分的；而将系统分为操作系统抽象层、通信协议层和应用功能层的分层机制是按照硬件距离来划分的。采用变化率作为分层标准使得划分后的功能能彼此独立地升级。

我们发现在许多应用中都结合使用了多种维度。例如，将应用分解为表现层、应用逻辑层和持久化数据层是一种结合了抽象和变化率两种维度的分层机制。用户界面通常比应用逻辑更容易发生变化，而应用逻辑又比数据格式——如关系数据库中的表结构——变化得快。不管应用采用什么样的分层机制，每一层都使用低层的功能来实现自己的功能。

关键问题之一是确定“合适”的层数。层数太少可能无法将系统中的不同问题充分地分离开以便独立升级。相反，层数太多又容易导致软件架构过于零碎而没有清晰的视图和范畴，从而非常难以改进。此外，定义的层数越多，端到端的控制流需要穿越的中间层就越多，可能会引入性能问题——特别是在层与层之间是远程的时候。

通常，层内自我完备且内聚的职责会被实现为一个单独的Domain Object，将同一层进一步划分为易于处理的部分，以便进行独立的开发和升级。

为每一层定义一个Explicit Interface (163)，将Domain Object的接口公开给其他层访问，并将Explicit Interface连接到实现相应功能的Encapsulated Implementation (181) 之上。这种分离关注点的方式减小了不同层之间的耦合，各层均依赖层接口，这样可以改进层内部的实现而不会对其他层产生影响，同时这种分离方式也为远程访问层的功能提供了可能性。Bridge (255) 或Object Adapter (256) 能帮助我们z同—层的Explicit Interface与Encapsulated Implementation分离开。

控制和数据流可以在分层系统中双向进行。例如，数据能在分层协议栈的相邻层之间交换，如TCP/IP或UDP/IP。然而，Layers定义了一个非循环的向下依赖关系：低层必须不依赖于高层提供的功能。这种设计避免了结构上的额外复杂性，同时能够在其他应用中使用低层功能，而与高层无关。因此，由下而上的控制流通常通过基于Observer (237) 回调基础设施来触发。低层可以通过由Command (240) 或Message (245) 方式实现的通知机制将数据和服务请求传递给高层，而不必依赖于高层接口的特定功能。

9.3 Model-View-Controller**

在将Domain Model (106) 转化为软件技术架构的时候，或设计Presentation-Abstraction-Control中某个代理的时候……我们必须考虑到应用的用户界面通常比其领域功能要变化得更频繁。



用户界面总倾向于发生变化，有些必须支持多种外观体验，另一些必须满足特定顾客的偏好。然而，更改用户界面必须不影响应用的核心功能，这部分通常是独立于外观的，变化也较少。

用户界面的修改应当比较容易，并且只局限在修改的界面部分。然而，可修改的用户界面不能以应用的服务质量的下降为代价，在任何时候它都必须能够显示当前的运算状态，并对状态的

变化做出及时的响应。更为复杂的是,对于支持多套外观体验的应用来说,每一套外观的变化率不同,从而需要进一步将不同的用户界面部分隔离开。

因此:将交互式应用划分为3个独立部分:处理、输入和输出。采用变化传播机制来确保这3部分的一致性(见图9-11)。

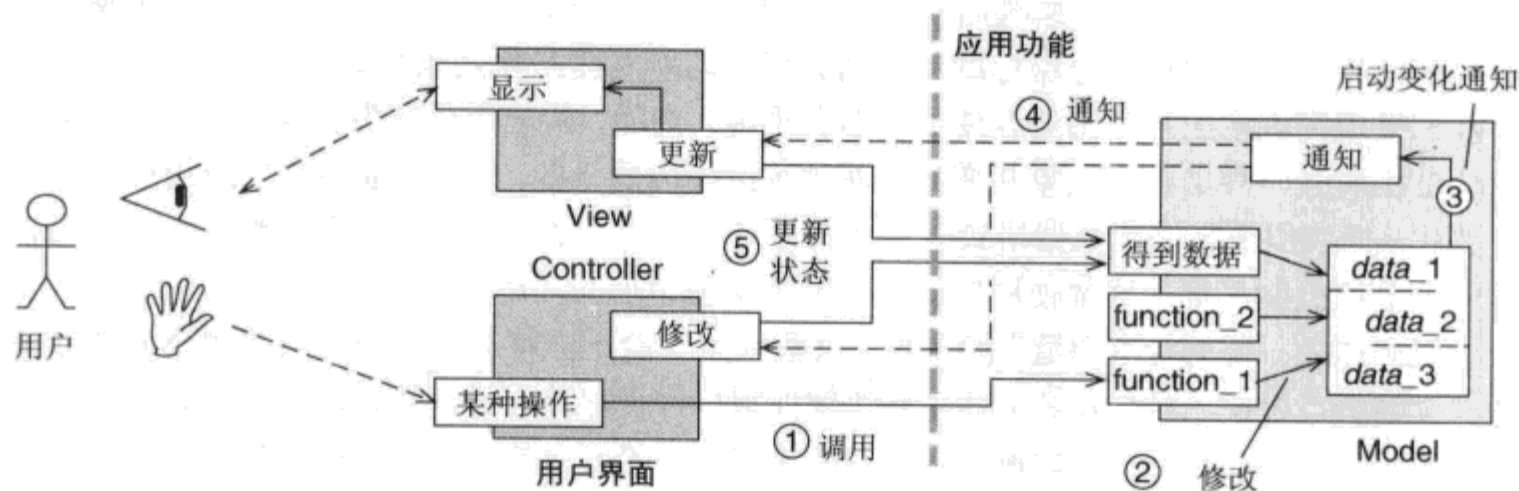


图 9-11

将应用的功能核心封装到Model中,其实现独立于具体的用户界面视感(look-and-feel)和机制。当Model中的某个方面需要通过应用的用户界面展示出来时,为其引入一个或多个自我完备的View。将各View和一系列独立的Controller关联起来,以接收用户输入并将其转换成对Model或相关View的请求。这样,用户就可以只通过Controller和应用交互。

将Model、View和Controller组件通过变化传播机制连接起来,当Model的状态发生变化,通知所有View和Controller做相应的改变,以便它们能通过Model的API完成对自身状态及时准确的更新。



Model-View-Controller的实现方式将不同变化率的应用职责分离开,从而支持它们独立地改进。

Model定义了交互式应用的功能核心,因此其内部结构严重依赖于应用特有的领域职责。通常Model被划分为一个或多个应用Domain Object (121),每个Domain Object负责一个自我完备的职责。Model的实现不应当依赖于具体的I/O数据格式或者View和Controller的API,从而避免了当用户界面改变时必须改变Model。

应用用户界面中所表现的每个内聚的信息段都封装在一个自我完备的View中,包括从Model中获取相关数据的功能、将数据转换成输出格式以及在用户界面中显示输出等。这种自我完备性允许View在互不影响及不改变Model的情况下自我改进。两种典型的View类型是Template View (200) 和Transform View (201)。Template View将Model中的信息输出到预定义的输出格式中。Transform View通过逐个输出从Model中获取的每个数据单元来形成输出界面。

系统中的每个View都与一个以上的Controller相关联以更新Model的状态。Controller通过相连的输入设备,如键盘或鼠标,接收输入并转换成对相关View或Model的请求。有3种常见的

Controller类型：与应用用户界面特定功能相关联的Controller；Page Controller (196)，处理用户界面中特定窗体或页面发出的请求；Front Controller (197)，处理对Model的所有请求。每个功能的Controller在Model支持大量功能时最适合。Page Controller对基于窗体或基于页面的用户界面比较适合，其中的每个窗体或页面提供一套相关的功能。当应用将功能公开给用户界面时，Front Controller最有用，其执行因每个特定请求的不同而不同，比如Web应用中的HTTP协议。

Controller发出的请求可以封装到Command (240) 对象中并传递给专门的Command Processor (199) 来执行。这样的设计允许Controller的改变对View和Model都是透明的。此外，它还支持将请求视为内在对象，反过来允许应用提供“内务”服务，如撤销/重做以及调度请求。

如果Controller不知道创建哪条具体命令，例如在工作流驱动的应用中，则可使用Application Controller (198)，它有助于避免对Model内部状态的依赖。在大多数应用中，同时有多个Controller处于活动状态，但每个用户输入只能由一个特定的Controller处理。Chain OF Responsibility (257) 可以将所有Controller连接起来，简化将特定输入分配给“正确的”Controller的过程。

通过Wrapper Facade (269) 来访问底层设备驱动的API和图形库时，View和Controller独立于系统平台，以及输入和输出设备。Data Transfer Objects (244) 有助于封装View和Controller从Model中取出的数据。

为了支持在Model、View和Controller间的高效协作而不破坏Model对用户界面的独立性，可以将它们通过Observer (237) 机制连接起来。Model作为主题，而View和Controller作为它的观察者。当Model的状态发生变化时，它通知所有已注册的View和Controller，View和Controller再从Model中获取相应数据来更新各自的状态。

9.4 Presentation-Abstraction-Control

在将Domain Model转化成软件技术架构时……我们必须常常考虑到应用的不同功能职责以及可能需要不同形式的用户界面。



人-机界面允许用户通过特定“方式”与应用交互，如窗体、菜单、对话框等。不过，某些应用最好通过不同的界面形式来操作其所提供的不同功能类型。

例如，在机器人控制系统中，定义任务的功能和在任务中控制移动的机器人功能需要不同的用户界面。而我们必须确保所有功能和界面构成一个内聚的系统。此外，对任何用户界面的改变不应当影响其相应功能的实现，也不应当影响其他功能及其相关的用户界面。类似地，某一特定功能实现上的变化也不应当影响用户界面和其他功能的实现。

因此，将交互式应用结构化，使其形成由多个解耦的代理构成的层次：一个顶层代理、几个中间层代理和许多底层代理。每个代理负责应用的一个特定功能并为其提供特定用户界面（见图9-12）。

底层代理实现了用户可与之交互的自我完备功能，如管理性任务、出错处理和数据操作。中间层代理协调多个相关的底层代理，例如用于呈现特定数据类型的所有View。顶层代理提供所有代理共享的核心功能，如访问数据库。

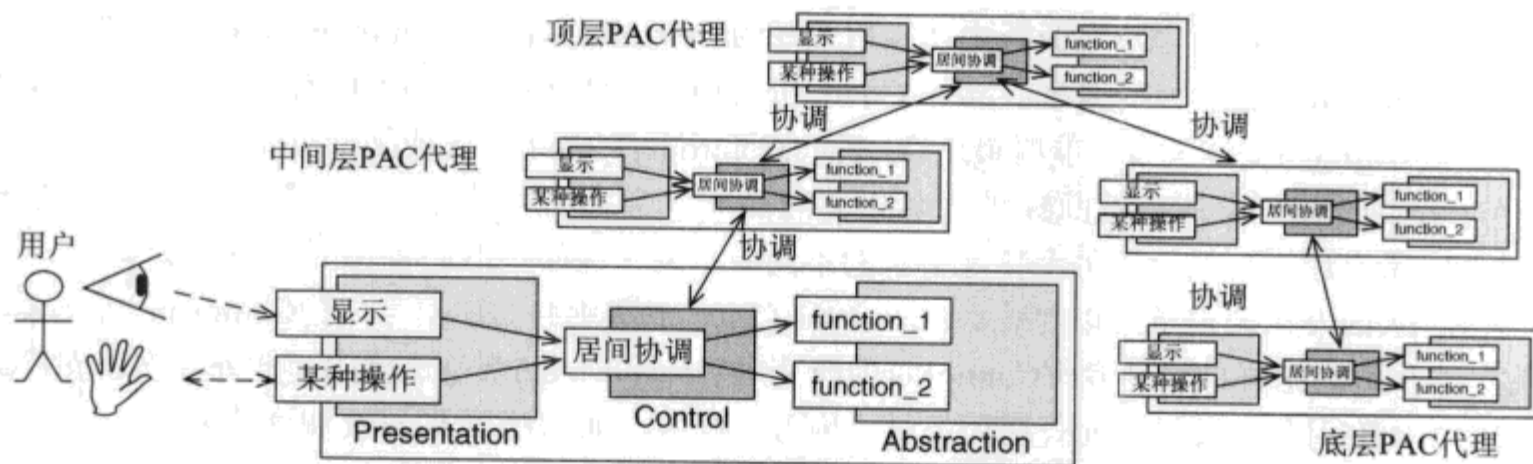


图 9-12

将每个代理分为3个部分：**Presentation**部分定义了代理的用户界面；**Abstraction**部分提供了代理相关的领域功能；**Control**部分将**Presentation**与**Abstraction**部分连接起来，并允许代理与其他代理互相通信。代理通过各自的**Control**部分形成层次化的结构。

用户通过**Presentation**部分与代理交互。用户对**Abstraction**部分相应功能的所有请求通过代理的**Control**部分进行协调。如果用户行为需要访问其他代理或者和其他代理进行协调，则将相应请求转发到这些代理的**Control**部分，沿着该结构向上或向上传输，并从**Control**部分传递给它们的**Abstraction**部分。



Presentation-Abstraction-Control架构有助于连接多个自我完备的子系统，甚至是连接整个应用，通过专门的人-机交互Model构成内聚的（分布式）系统。这种方式的不足之处在于其复杂性：必须提供多个用户界面，而且当控制流分散在多个子系统中，当需要在相关的用户界面中做出反应或更新View时，必须对某个用户界面触发的行为进行仔细明确地协调。因此，**Presentation-Abstraction-Control**架构只有在软件系统不采用单一用户界面模式实现时才有实用价值。

为实现**Presentation-Abstraction-Control (PAC)**架构，必须识别出应用向用户提供的所有自我完备的职责。再将每个职责封装在一个单独的底层代理中。如果几个代理共享某种功能或需要协调，那么需要将该（协调）功能整理到中间层代理中。在PAC架构中可以有多层的中间层代理。所有代理共享的功能由顶层代理提供。这种分离方式支持代理的独立修改而不影响其他代理，同时允许每个代理提供自己的用户界面。为每个代理提供Model-View-Controller (109) 架构：**Abstraction**部分相当于MVC中的Model并被划分成Domain Object (121)，**Presentation**部分相当于View和Controller。因此，改变代理的接口会影响其实现。

代理的**Abstraction**部分和**Presentation**部分通过Control组件来分离，Control组件作为Mediator (239) 完成两方面的职责。首先，它应当将代理的**Presentation**部分所发出的所有用户请求转发给**Abstraction**部分的适当功能。它还必须将**Abstraction**部分发出的所有的变更通知传递给**Presentation**部分的View。其次，Control部分必须协调代理间的合作。如果某个代理收到的用户请求不能被自己单独处理，那么Control部分会将该请求，连同相关输入数据，一起转发给适当的高层或低层代理。结果用同样的方式传回，但是方向正好相反。代理的Control部分可以从其他代理

的Control部分接收请求和数据。要转发的请求可以封装在Command (240) 对象以及Data Transfer Objects (244) 的数据中。Control部分是代理间松散耦合的关键：如果代理的Abstraction部分发生变化，仅限于影响其他代理的Control部分。

为了使代理之间保持一致性，可通过Observer (237) 方式将它们连接起来。如果代理依赖于与其相连的高层或底层代理的状态，可以将它的Control部分作为订阅者注册到相应代理的Control部分上，而相应代理的Control部分就相当于Observer模式中的主题部分(subject)。只要在这些“主题”代理状态发生变化时，其Control部分才会通知所有“观察者”代理，从而使后者做出响应，更新自身状态。

9.5 Microkernel**

在将Domain Model (106) 转化成软件技术架构的时候……我们在设计中必须支持在不同部署情形下，功能的可伸缩性和适应性。



有些应用存在多个版本。每个版本向用户提供不同的功能集，或者是和其他版本在特定方面有所不同，比如用户界面。尽管它们存在差异，应用的所有版本都应当基于一个公共的架构和功能核心。

这样做的目的是为了应用在不同版本之间出现架构漂移，以降低共享功能的开发和维护成本。此外，从应用的一个版本升级到另一个版本，如增加或删除一些特性，或者改变其实现，应当不需要对系统做出改动，或只需要做很少的改动。类似地，应当很容易为应用特定版本提供不同的用户界面，或者让该版本运行在不同的平台上，允许客户端在特定环境中以最合适的方式使用。

因此，通过“即插即用”机制来扩展一个公共且最小的内核来为应用构造不同的版本（见图9-13）。

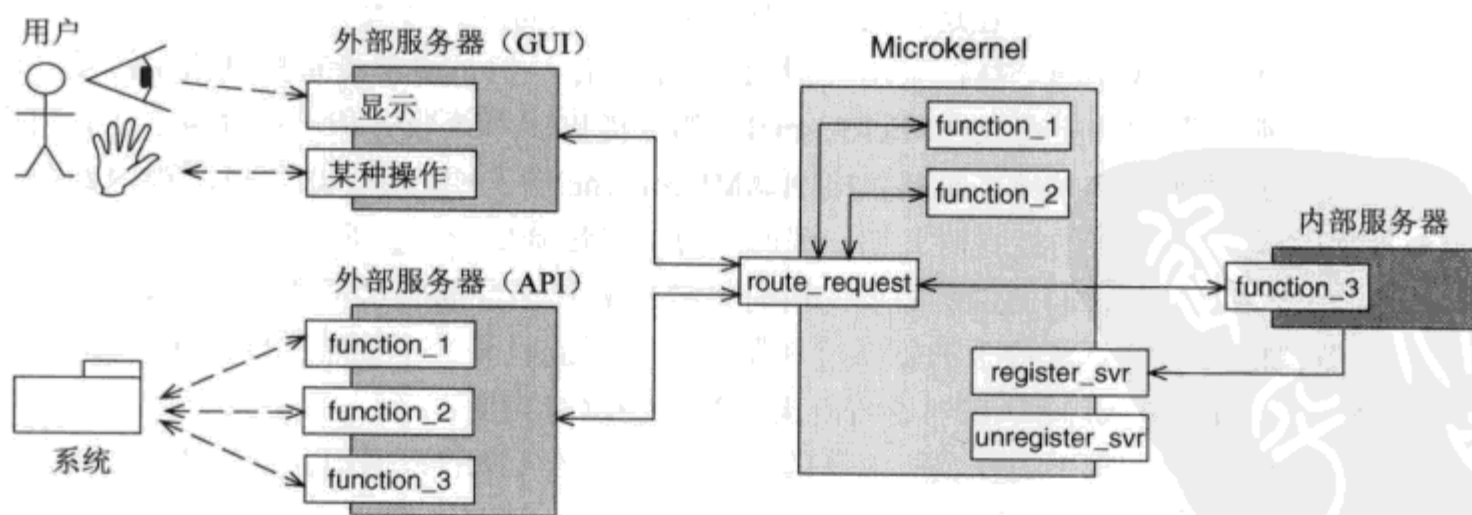


图 9-13

Microkernel实现应用的所有版本共享的功能，并为集成某个版本特有的功能提供基础设施。内部服务器实现自我完备的特定版本的功能是；外部服务器实现特定版本的用户界面或API。要

配置应用的特定版本，我们只需要将相应的内部服务器和Microkernel连接起来，并提供合适的外部服务器来访问其功能。因此，应用的所有版本共享一个公共的功能和基础设施内核，但却可以提供不同的功能集和外观体验。

客户端，无论是人还是其他软件系统，完全通过外部服务器提供的接口或API访问Microkernel的功能，外部服务器负责将所有接收到的请求转发给Microkernel。如果Microkernel自身实现了所请求的功能，则执行该功能，否则将请求转发给相应的内部服务器。相应的返回结果也会通过外部服务器显示或传递给客户端。



Microkernel架构确保应用的每个版本可以完全按照其目的进行裁剪。用户或客户端系统只得到其要求的功能和外观体验，而不会为他们所不需要的特性引入任何开销。一般来说，改进某一特定版本以包含新的或不同功能和特性，“只”需要重新配置合适的内部和外部服务器，Microkernel自身不受这种升级的影响。已存在的内部和外部服务器以及应用的其他版本同样不受影响。此外，Microkernel架构降低了开发和维护整个应用系列中所有成员的成本，每个服务、用户界面和API只需要实现一次。

Microkernel的内部结构通常是基于Layers (108) 的。最底层由底层系统平台抽象出来，因此支持所有高层的可移植性。第二层实现Microkernel所依赖的基础设施功能，如资源管理。上层负责所有应用版本共享的领域功能。最顶层包括配置内部服务器和Microkernel的机制，并且将外部服务器的请求转发给感兴趣的接收方。

Microkernel中每个特定的自完备的功能和职责可以实现为一个Domain Object (121)，以支持独立的实现和改进。Microkernel的路由转发功能通常采用Mediator (239) 来实现，它通过统一的接口接收请求并将这些请求分配给Microkernel或内部服务器中的相应领域功能。为了降低资源的占用，特别是内存的使用，路由转发层可以使用Component Configurator (289) 或Object Manager (291) 在需要的时候加载内部服务器，用完后从内存中卸载，并控制其生命周期。这种设计还支持对应用的特定版本进行升级，在运行时动态加入新的、不同的或修改后的功能。

内部服务器和Microkernel一样，采用类似的Layers设计，但是通常没有提供路由转发一层。此外，如果内部服务器的功能建立在Microkernel中的层提供的系统服务和平台抽象之上，它们可以避免自己实现这些服务和抽象，而是直接回调Microkernel中的相应层。这可以减少内部服务器的使用，但回调会引入额外的运行时开销。因此，为了尽量降低分布式系统中的网络流量，并增强内部服务器的性能，为它们提供所需要的所有系统服务和平台抽象也可能是有益的。

外部服务器的设计严重依赖于其复杂度和用途。它可能只是一个简单的Object Adapter(256) 以将应用公开的API映射成内部API，也可能提供复杂的用户界面。

在外部服务器、Microkernel以及所配置的内部服务器间交换的数据与具体应用有很大关系，它们可以封装在Data Transfer Object (244) 中。

9.6 Reflection*

在将Domain Model转化成软件技术架构的时候……有时我们提供的设计必须能够应对未来

的改进，或者需要集成一些无法预料的变化。



对于那些长期存在的应用来说它们的可持续架构关键之处就在于要能够支持变化：随着时间流逝它们必须能够对不断变化和改进的技术、需求以及平台做出响应。然而，我们很难预见应用中哪些部分会发生变化，以及何时对特定的变化请求做出响应。

更为复杂的是，这种对变化的需求可能在任何时候出现，特别是在产品已投入使用的环境下。变化的规模也大不相同，从算法的本地调整到分布式基础架构的根本性改变。尽管应用通常都会经历多次变化，但是对于某些特定的变化来说，我们不希望给维护人员带来负担，而且，应当存在一种统一的机制来支持不同类型的变化。

因此，将应用的结构、行为以及状态方面的属性和变化因素具体化到一套元对象中。通过两层架构将元对象和核心应用逻辑分开，元数据层包含元对象，基础层包含应用逻辑（见图9-14）。

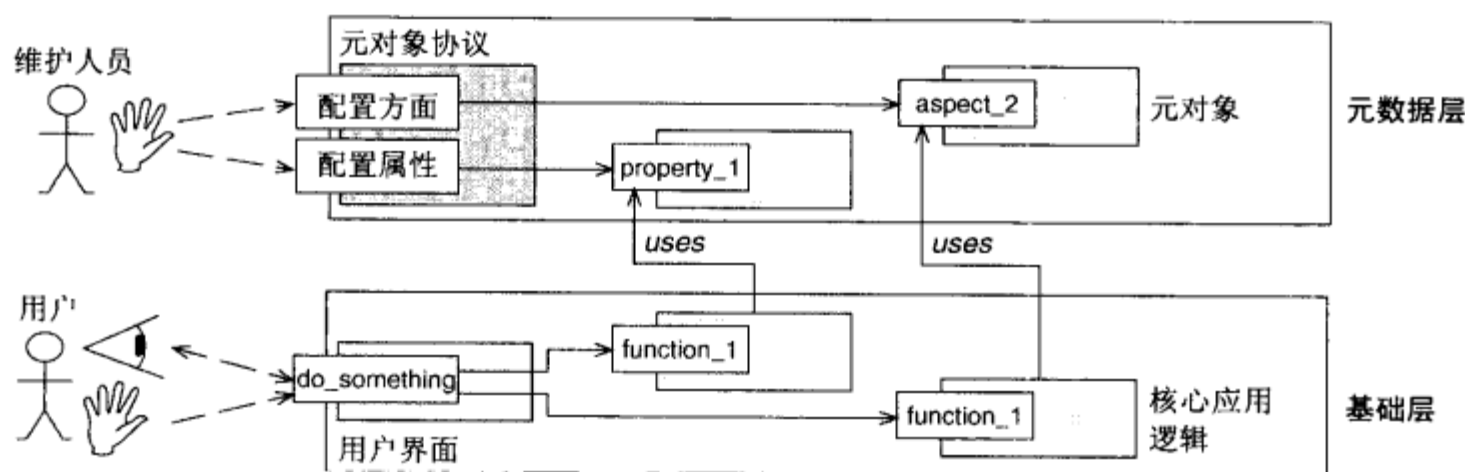


图 9-14

为元数据层（meta level）提供元对象协议，作为专门的接口供管理员、维护人员甚至是其他系统使用，在应用的监督控制下动态配置或修改所有元对象。将基础层与元数据层连接起来，以便基础层在执行具体行为或访问可能发生变化的状态之前查阅适当的元对象。



Reflection在软件架构中很大程度地支持了运行时的灵活性。它使得软件系统的几乎任何信息都可以被访问，任何方面都能被改变。因此，某些编程语言直接支持特定形式的Reflection，如Java的java.lang.reflect包以及C#中的System.Reflection命名空间。然而，必须注意Reflection架构的重量级机制只有在具有相匹配的重量级的灵活性需求时才有意义。

要实现Reflection架构，首先为应用制定一个稳定的设计，完全不考虑其灵活性：稳定性是灵活性的关键[Bus03]。通常，应用的每个完备的职责都封装在Domain Object(121)中，它们一起构成了Reflection架构的基础层。

使用合适的方法，如Open Implementation Analysis and Design（开放的实现分析和设计）[KLLM95]、Commonality/Variability Analysis（通用性/可变性分析）[Cope98]或Feature Modeling（特性建模）[CzEi02]——找出应用在结构和行为上可变的方面。通常情况下，行为上的变化包括

应用功能的算法、Domain Object的生命周期管理、事务协议、IPC机制以及安全策略和故障处理。还可能需要在系统加入全新的行为或移除已有行为。

结构上的变化因素包括应用的线程或进程Model、Domain Object的进程和线程部署，或者是系统的类型结构。此外，还需要确定可以影响应用行为的全部系统级信息、属性和全局的状态，如Domain Object所提供接口、内部结构以及是否可持久化等运行时类型信息。

将上述分析中找到的每个可变的行和结构方面、系统属性和状态实现为单独的元对象，并将所有元对象放在Reflection架构的元数据层。有了这样一种严格的封装机制，我们就可以显式地访问这些方面，并能在任何时间对其进行修改或调换。对元对象的修改不会导致应用的基础层实现受到影响。

将基础层中每个Domain Object的实现开放，使其可以查阅元对象并访问封装在元数据层中的各个方面。这样，对元对象的改变就能立刻影响基础层接下来的行为。

为了支持在运行时创建、配置、交换或销毁元对象，我们引入元对象协议作为管理元数据层的单一接口。为了完成这些操作，我们需要管理元对象生命周期的基础设施：Abstract Factory (311) 和Builder (312) 可以用来实现创建和销毁元对象；Component Configurator (289) 和Object Manager (291) 可以用来控制元对象生命周期特定步骤的执行。类似的设计使得我们可以在反射型应用及其元数据层投入使用之后，集成新开发的元对象。Introspective Interface (166) 和Dynamic Invocation Interface (167) 支持应用的外部客户端，如测试框架或对象浏览器，获取基础层的Domain Object信息而不必依赖其内部结构，以及调用其方法而不必使用其功能接口。

元对象协议和Reflection架构的两层结构结合起来就是如何开/闭原则[Mey97]的一个具体示例。元对象协议将软件改进的复杂性隐藏在“更简单”的接口之后，使其更加容易、统一和动态化，同时允许反射型的应用监督自身的改进以尽量减少不受控的改变。将Reflection架构分为基础层和元数据层使得应用中的不变因素和可变因素分离开：可以在管理元对象的同时不影响基础层Domain Object的内部设计和实现。

9.7 Pipes and Filters**

在将Domain Model (106) 转化成软件技术架构的时候……有时我们必须提供一个适合的设计来处理数据流。



有些应用处理的对象是数据流，输入数据流会被按步骤转换成输出数据流。通常情况下，使用我们所熟悉的请求/响应机制来确定这种类型应用的结构往往是不可行的。我们必须为它们制定适当的数据流模型。

为数据流驱动的应用建模会在开发和运行上引入不小的挑战。首先，应用的各部分应当与数据流上离散的、可区分的行为相对应。其次，有些应用场景要求明确访问有意义的中间结果。第三，选取的数据流模型应允许应用采用增量方式读出、处理并写入数据流，而非全部顺序访问所有数据，这样才能保证吞吐量最大。最后，但并非最不重要的是，长时间的处理活动必须不成为性能上的瓶颈。

因此，将应用的任务划分为几个自我完备的数据处理步骤并将这些步骤通过中间数据缓存连接到—个数据处理管道中（见图9-15）。

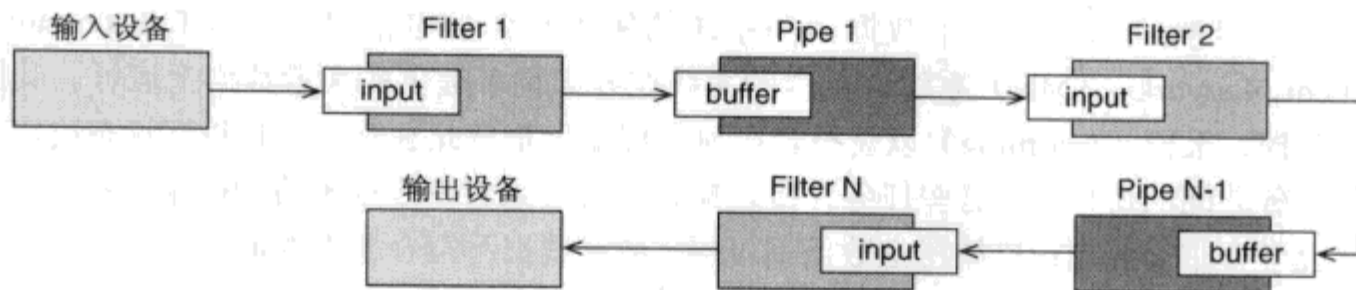


图 9-15

将每一个处理步骤实现为单独的过滤器组件，不断地消耗和分发数据，同时将所有过滤器链接起来构成整个应用的主数据流模型。在数据处理管道中，一个过滤器产生的数据由后续的过滤器处理。相邻的过滤器通过管道分隔开，管道缓存过滤器之间交换的数据。



Pipes and Filters 架构将不同的数据处理步骤分离开，从而使它们可以彼此独立地改进并支持增量数据处理方式。

在 Pipes and Filters 架构中，过滤器是领域相关运算的基本单位。每个过滤器可以实现为一个 Domain Object (121) 以表示一个特定的自我完备数据处理过程。采用并发性 Domain Object 实现方式的过滤器能实现增量和并发性的数据处理，从而增强了 Pipes and Filters 方式的性能和吞吐量。如果某个过滤器要进行长时间活动，可以考虑在处理链中集成多个并行的过滤器实例。这样的配置可以进一步提高系统性能和吞吐量，因为某些过滤器的实例可以在其他过滤器处理先前的数据流时开始处理新的数据流。

在 Pipes and Filters 架构中管道是数据交换和协调的媒介。每个管道在过滤链中实现了缓存和传送数据的 Domain Object，产生数据的过滤器将数据写入管道，同时消耗数据的过滤器从管道中获取其输入数据。管道的集成分离了相邻的过滤器，这样一来，过滤器间就可以彼此独立运行，从而使得它们各自的运行性能达到最佳。

在单进程的 Pipes and Filters 实现中，管道通常以队列的形式实现。采用并发性 Domain Object 实现的管道允许增量和并发性的数据处理，正如并发性过滤器一样。在分布式实现中，管道采用一些如 Messaging (129) 机制的方式在远程的过滤器之间传递数据流。将管道实现为 Domain Object 可以使得过滤器不需要知道其特定实现，而且还能透明地替换不同的实现方式。这种设计支持分布式 Pipes and Filters 架构中的灵活（重新）部署。Message (245) 有助于封装管道中传送的数据流。

9.8 Shared Repository**

在将 Domain Model (106) 转化成软件技术架构的时候……有时我们必须为应用提供一个良好的设计用来操作某些共享的数据集合，或者通过这些共享的数据集合来协调各部分之间的合作。



某些应用本质上是数据驱动的，组件之间的交互并不遵循特定的业务流程，而是依赖于其操

作的数。也许我们很难找到一种有效的方法将这种应用的各个部分连接起来，但我们还是要保证它们之间的交互是可控的。

网络管理控制应用就是一个数据驱动系统的例子，比如电信管理网（Telecommunication Management Network, TMN）系统。这样的系统在外设提供的大量数据上运行。其核心功能，比如监控、报警（Alarming）以及报表在很大程度上是彼此独立的，其数据状态决定了控制流和这些任务之间的协作。将这些任务直接连接起来会导致将特定业务流程硬编码在应用中，这在某些情况下会带来问题：比如特定数据不可用，或者数据不符合要求的质量，或者数据不处于要求的状态。然而，我们需要一个贯穿整个应用的内聚运算状态。

因此，将所有数据维护在Central Repository（中心仓库）中，为数据驱动应用的所有功能组件所共享，同时让数据的可用性、质量及状态来触发和协调应用逻辑的控制流（见图9-16）。

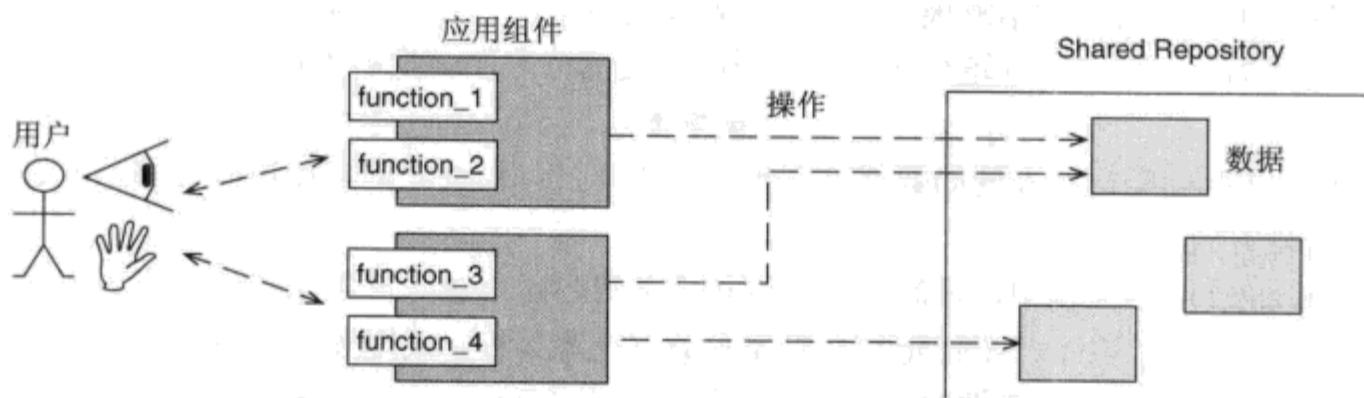


图 9-16

组件直接工作在Shared Repository维护的数据之上，这样其他组件就可以在数据变化时作出反应。如果某个组件创建了新的数据，或者应用从环境中接收到新的数据，那么也要将数据存放到Shared Repository中，以便其他组件访问。



Shared Repository架构能将数据驱动的控制流和应用的功能集成起来，形成内聚的软件系统。它还可以将操作同一份数据的应用集成在一起，而不要求这些应用共享或参与公共的业务流程。通过共享数据的状态来协调组件可能会引入性能和可伸缩性方面的瓶颈，特别是当多个并发性的组件需要独占访问同一份数据，而要求串行访问时。

对于数据驱动的应用来说，Shared Repository是其中央控制协调单元和数据访问点。它既可以简单到只是内存中的数据集，也可能像外部数据仓库那么复杂，需要通过Database Access Layer (318) 来访问。如果Shared Repository实现为Domain Object (121)，应用组件可以不关心其内部实现，进而能够透明地被修改或替换。Data Transfer Objects (244) 有助于封装Shared Repository和应用组件之间传递的数据。

Shared Repository维护的数据经常封装为Managed Object——隐藏了具体数据结构的细节，并提供有意义的访问和修改操作的Domain Objects。Managed Object允许应用组件使用特定数据而不需要依赖其具体表现形式，且在不影响使用该数据组件的情况下更改其表现形式。Managed Object还可以通过相应的质量属性来表示其数据的质量，比如数据是最新的、过时的、不确定的或损坏

的。组件可以使用该信息来控制对该数据的特定处理。

一般来说，访问Shared Repository及其Managed Object必须是同步的，因为应用的多个组件可能同时访问数据。在大部分配置下，这种同步会发生在Managed Object的层次中，从而为数据驱动型应用提供了最大的潜在并发性。可以为Managed Object提供一个Thread-Safe Interface (224) 来强制实现Managed Object接口的同步。如果只有一小部分的方法处于关键段中，通过Strategized Locking (226) 实现同步也是一种可行的备选方案。采用Monitor Object (214) 实现Managed Object以对多个组件同时访问Managed Object提供协作式并发性控制。

许多Shared Repository在仓库数据变化时提供了通知应用组件的机制。例如，可能插入了新数据、修改了已有数据或删除了部分数据。组件因此可以在仓库数据变化时及时做出反应。在大多数情况，变化通知机制采用Observer (237) 方式实现：Shared Repository作为主题 (subject)，应用的组件作为观察者(observer)。类似地，Managed Object也可以提供基于Observer的变化传播机制，从而能够通知组件特定值发生变化。

这两种选项中那一个更适合数据驱动的应用要看具体的职责。通常的考虑是在简单性和粒度之间进行折衷：提供Shared Repository级的变化通知易于实现，但是当观察者对所报告的变化不感兴趣时会导致额外的开销。与之相反，提供Managed Object级的通知机制避免了不必要的通知和数据传输，但是更加复杂。在Shared Repository的全部数据上运行的应用组件越多，提供Shared Repository级的通知机制就越可行；而当组件更倾向于有选择地访问Managed Object时，提供Managed Object级的通知机制显然更合适。

应用的数据驱动服务组件通常实现为Domain Object，通过访问和操作Shared Repository中的数据来实现特定功能。组件之间的合作完全在数据层实现，在某个Managed Object改变Shared Repository中的数据状态、插入或者删除数据时通知其他组件。

9.9 Blackboard**

在将Domain Model (106) 转化为软件技术架构的时候……有时候我们提供的设计必须适合那些没有已知的确定解决方案的应用。



对于有些任务来说，没有已知的确定性解决算法，只有近似的或不确定的知识可用。尽管缺少合适的算法支持，反复试验的技术也有可能足够成功，而且我们必须为这类任务开发有生产价值的应用。

这种系统的例子包括语音识别、基于声纳信号的海底探测，以及由X射线数据得出的蛋白质分子结构推论。这些任务必须解决几个有挑战性的困难：输入数据经常是模糊或不准确的，必须探索出得到解决方案的途径，每一个处理步骤可能产生交替的结果，而且经常没有已知的优化方案。此外，在合理的时间内计算出有价值的方案非常重要。

因此，通过将多个具有确定性解决方案算法的更小组件结合起来，使用启发式的运算来解决这类任务以逐渐改善过渡性的假定解决方案（见图9-17）。

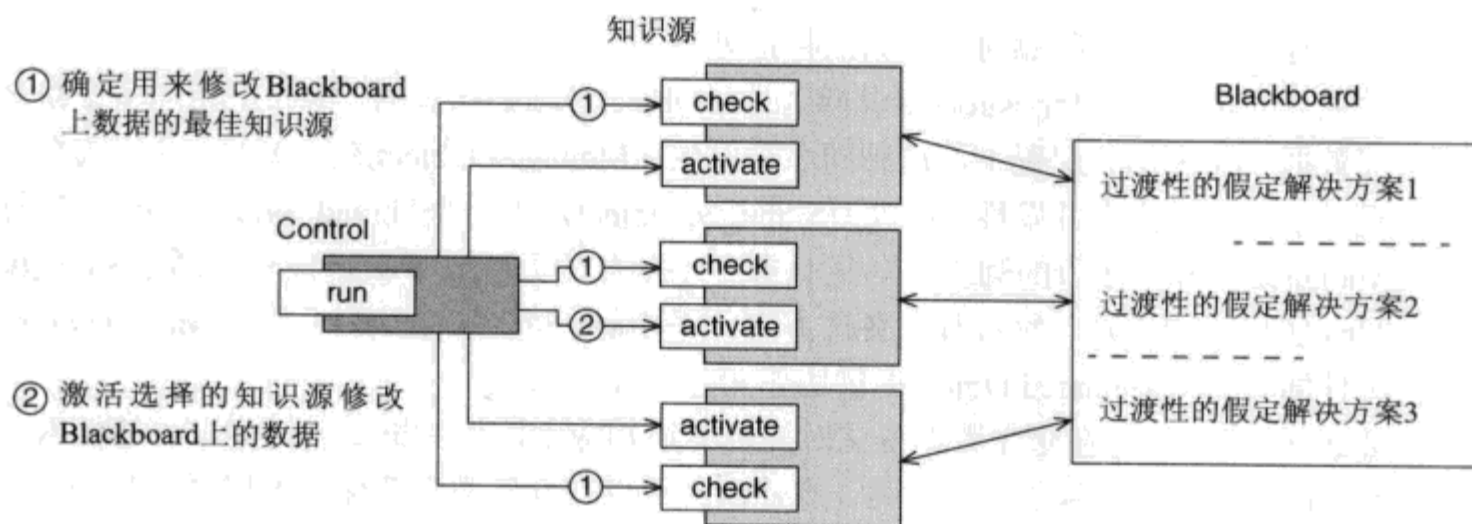


图 9-17

将系统中的全部任务划分为一系列较小的、自我完备的子任务，这些子任务都有已知的确定性解决方案算法，再为每一个子任务的职责分配给一个独立的知识源。为了允许知识源可以按照任意顺序彼此独立地执行，可以让它们通过非确定性的数据驱动机制来合作。通过使用共享数据仓库和Blackboard，知识源可以判断它们是否有输入数据可用、处理输入数据，以及交付结果，这些结果又可能形成系统中其他知识源的输入。

使用一个控制组件来协调计算，采用尝试性启发式方法选择并激活适当的知识源，如果Blackboard上的数据不能作为最终数据，则重复上述行为，否则停止计算。这种策略是通过对部分结果的增量式改进和对不同假设的估算来获得问题的解，而非通过确定的算法。



Blackboard架构有助于构造那些必须解决不确定性任务的软件系统，这些任务通常基于模糊的、假设的或者不完整的知识和数据。它还有助于为这类任务发现和优化在很大程度上具备确定性的解决方案。另一方面，我们不能保证基于Blackboard的系统一定能产生有用的结果。此外，这种基于启发式的计算方案对于那些要求预见出结果质量和结果产生时间的情况并不可行。

要实现Blackboard系统，首先应当将待解决的任务分解。系统需要接收什么样的输入？它需要产生什么样的输出？有什么样的潜在解决方案路径以及能得到解决方案什么样的中间结果？有助于得到解决方案（路径）的已知算法是什么？每个算法能处理什么样的输入或中间结果？每个算法能交付的中间或最终结果是什么？在这些分析的基础上，为任务解决方案相关的每个算法定义自我完备的独立可执行的知识源。这种独立性允许知识源的执行顺序是任意的——这是采用启发式方案策略的必要前提。为了使启发式方法能够确定某一特定的执行顺序，我们将每一个知识源分成两个单独的部分。条件部分（condition part）通过审核Blackboard中已写的数据来检查该知识源是否可能对运算过程做出贡献。行动部分（action part）实现知识源的功能：从Blackboard上读取一个或多个输入数据，处理数据，并将一个或多个输出数据写回Blackboard。如果行动部分发现数据对任务的解决已经没有作用的话，它也可以从Blackboard中擦除数据。通常，每个知识源采用一个Domain Object (121) 来实现，从而在应用的整个任务有更多知识可用时能支持它的独立改进和优化。

Blackboard作为一个数据仓库来维护知识源产生的所有部分或最终的结果。它可以设计为一个内存中的数据集合，或者通过Database Access Layer (318) 访问的外部数据仓库。如果将Blackboard实现为一个Domain Object，可以对知识源隐藏其具体实现，采用透明的替换和修改。Data Transfer Object (244) 有助于封装在Blackboard和知识源之间传递的数据。

控制组件实现Blackboard系统的启发式算法策略。它首先读取系统输入并保存到Blackboard中，然后循环执行以下三步：第一步调用所有知识源的条件部分来判断在当前运算状态下是否有用；第二步使用启发式算法来分析条件部分返回结果以决定哪一个知识源对运算进程最有效；最后激活选中知识源的行动部分，修改Blackboard中的内容。一旦该知识源结束执行，继续重复执行该循环。

如果Blackboard中包含了一个正确的最终结果则整个循环结束，或者当没有知识源能改进任意中间假设方案的质量时也结束循环。将控制组件实现为Domain Object以允许透明地修改和改进所选的启发式算法，这时不会对Blackboard结构中具体的知识源和Blackboard产生任何影响。

9.10 Domain Object**

在实现Domain Model (106)，或者在技术架构中采用Layers (108)、Model-View-Controller (109)、Presentation-Abstraction-Control (111)、Microkernel (113)、Reflection (114)、Pipes and Filters (116)、Shared Repository (117) 或Blackboard (119) 的时候……所有这些设计工作的关键点在于将自我完备且内聚的应用职责彼此分开。



构成软件系统的各部分之间经常表现出各种协作和包含关系。然而，若不仔细地实现这种互相关联的功能，则最终产生的设计在结构上可能会过于复杂。

优秀软件设计的关键属性是分离不同的关注点。软件系统不同部分解耦合做得越好，就越能够进行独立的开发和改进。各部分之间的关联越少，软件架构的复杂度就越低。各部分之间耦合越松，在计算机网络中就能部署得越好，或者能更好地组成更大的系统。换句话说，软件系统的合理划分能避免架构上的碎片，从而使得开发人员可以更好地维护、改进和分析。尽管我们需要清楚地分离关注点，但对某些关键运行质量来说，如性能、出错处理以及安全性，软件系统的实现以及不同部分之间的协作必须是高效的。

因此，将应用中所有明确的功能封装为一个自我完备的构造块——Domain Object中（见图9-18）。

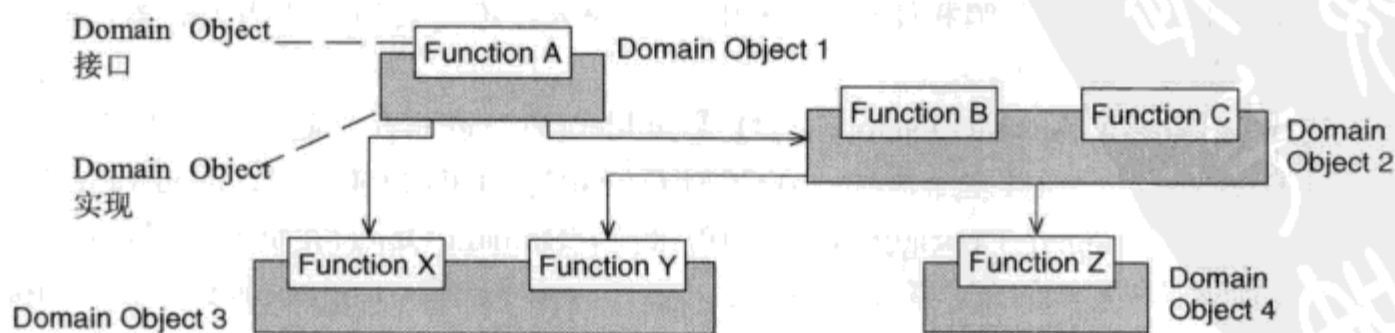


图 9-18

为所有Domain Object提供一个和实现相分离的接口，每个Domain Object程序只使用这些接口来访问其他Domain Object。



Domain Object将应用中不同的功能职责分开，每个功能都被很好地封装起来，从而可以独立改进。基于一个或多个粒度上的标准，我们可以将应用职责划分为Domain Object的特定形式。Application Service[ACM01]是一个Domain Object，它封装了自我完备且完整的业务功能或应用的基础方面，如银行、航班预定或日志服务[Kaye03]。Component[VSW02]也是Domain Object，用于封装功能性的构造块，如所得税计算或现金兑换，或者是领域实体，如银行账户或用户。Value Object[PPR][Fow03a]、Copied Value (230)以及Immutable Value (231)是一些小的Domain Object，其标识由自身的状态而不是类型来决定，如日期、货币交换率以及金钱的数额。一个Domain Object还可以包含其他相同或更小粒度的Domain Object。例如，服务通常由使用值对象的组件所创建。

将每个Domain Object分为Explicit Interface (163)和Encapsulated Implementation (181)，Explicit Interface部分向外导出其功能，Encapsulated Implementation部分实现其功能。这种接口和实现的分离使得Domain Object间的耦合最小——每个Domain Object只依赖其他Domain Object的接口，而不依赖其具体实现。这样在实现和改进Domain Object的时候，对其他的Domain Object就不会产生影响。Domain Object的Explicit Interface能够对其关键操作属性进行约定，如其他Domain Object可以依赖的错误行为、安全方面等。

将Domain Object的Explicit Interface和Encapsulated Implementation连接起来有几种可行的方式。如Java和C#在其语言的核心中支持Explicit Interface的概念，而且类（Encapsulated Implementation）可以直接实现它们。其他的静态类型语言，如C++，其Explicit Interface可以由抽象基类表示，具体的实现由该抽象基类派生出来。

Bridge (255)或Object Adapter (256)可以将Domain Object的Explicit Interface与Encapsulated Implementation明确分离开，二者可以独立地变动。Domain Object中Explicit Interface与Encapsulated Implementation的分离程度取决于其粒度和变化的相似性。Domain Object越小，例如实现Value Object或Immutable Value，严格分离带来的好处就越少。类似地，Encapsulated Implementation改变得越频繁，Explicit Interface就应当被分离得越彻底。

Explicit Interface还能实现远程访问Domain Object。不过，必须注意这种远程通常只对“大的”Domain Object可行，如服务或粗粒度的组件，对于“小的”Domain Object则不适用，如值对象。Domain Object越小，其网络开销和计算时间的比例就越大，进而会损害到相关的运行质量因素，如性能、可用性以及可伸缩性。

Domain Object经常和Abstract Factory (311)或Builder (312)联系在一起，以允许客户端透明访问其Explicit Interface及管理其生命周期。在CCM[OMG02]、EJB[MaHa99]、.Net[Ram02]等平台上，由Declarative Component Configuration (270)来控制Domain Object并负责其指定宿主环境应当如何处理生命周期和资源管理，以及事务、日志等其他技术因素。Component Configurator (289)可以帮助我们在运行时载入、替换、（重新）配置以及卸载Domain Object。



广东肇庆变电站的直流高压传输线路。
中国西门子出版社图片© Siemens AG

如果只考虑应用、主操作系统（host operating system）和网络，我们将很难满足复杂的分布式系统的要求，比如可伸缩性、可靠性等。应用应该只关注“业务逻辑”，而不是繁杂的底层基础实现（plumbing），操作系统和网络应该分别关注终端系统（endsystem）的资源管理和通信协议的处理。为满足其他关键方面的需求，本章介绍了12个与中间件相关的模式。所谓中间件，就是分布式基础设施软件，它可以为应用屏蔽来自于操作系统和网络的那些固有的或难以预料的复杂性。

有两种趋势会影响我们思考和创建分布式系统的方式[ScSc01]。

- 信息技术正一步步地走向大众，硬件和软件正在以相对可预测的速度变得更强大、更便宜和更卓越。硬件（比如CPU和存储设备）、网络元素（比如IP路由器和WiFi设备）大众化的过程，已经经历了几十年的发展。最近，得益于Java、C#和C++这些面向对象语言，以及Linux、Windows和Java虚拟机等商用套装（commercial-of-the-shelf）操作环境的日臻

成熟，软件本身也正在走向大众化。

- 人们对面向服务的软件范型的接受程度正在持续增长，为了满足各种需求，人们通过整合多个独立的服务，来建立分布式系统。这些服务是通过不同形式的网络协议连接到一起的。这些互联自然会跨越不同类型的应用：小到微型的、紧密耦合的应用（比如防锁死制动系统），大到庞杂的、松耦合的系统（比如因特网和万维网）。

这两种趋势互相影响，由此衍生出了新的架构概念和服务，并且使中间件（比如MQ Series、CORBA、Enterprise Java Bean、DDS和Web Service）层变得明晰起来。中间件是一种分布式基础架构软件，它位于应用与底层的操作系统、网络协议栈和硬件之间。它主要的角色是沟通应用与底层的硬件和软件基础设施，以及协调应用的各部分，将它们连接起来并实现互操作。中间件还使得整合由不同的技术供应商开发的组件成为可能，而且降低了整合的难度。

如果能够正确地实现和应用中间件，那么它可以解决很多第2章中谈及的挑战。例如，它为开发者屏蔽了大量的底层平台细节，比如Socket级别的网络程序设计，因此开发者可以更关注于应用的业务逻辑需求。中间件还有助于将软件生命周期的开销进行分解。由于中间件的存在，开发者可以通过重用框架和服务，以便在网络化的环境中高效地运作，而不必完全由自己从零开始实现，从而减轻了开发前期对于专业知识需求的压力。

开发一种通信中间件，以解决第2章中列出的各种挑战，这是一项复杂、耗时的工作。幸运的是，你几乎很少需要自己去设计和实现。目前有大量的通信中间件标准，以及现成的商业平台可用，比如CORBA [OMG04a]、.NET Remoting [Ram02]、Microsoft通信框架和JMS[HBS+02]，这些都已成功地应用于大量分布式系统中了。

有如此众多不同的标准和产品共存，其麻烦之处自然在于你需要为项目和系统考虑更多的选择。正确选择出“最佳的”通信范式、中间件标准和产品，依赖于很多因素，包括价格、支持、质量和待开发系统的需求。在分布式系统中，很少能有一种中间件解决方案对所有应用都是最合适的。

为了提高在一个分布式应用中的生产率，开发者还必须正确使用所选的中间件产品。有很多项目[Bus03]，其失败的原因都是没有充分理解通信范式、缺乏对中间件关键结构和行为的了解。因此，在选择和使用特定的通信中间件时，必须深思熟虑，并给出明确的决策。

我们将这一章包含在分布式系统的模式语言中，主要目的是帮助你理解不同的通信中间件原理及其内部设计。你会了解到每种方式的基本特性和它们的优缺点。有了这些知识准备，你就能在自己的分布式系统中从容地为应用选择出正确的通信范式和中间件了。

尽管不同中间件的技术细节各有不同，但是它们基本上都遵循3种不同的通信风格中的一种或多种：消息、发布/订阅和远程方法调用。在本章中，它们分别反映在下面的3种入口点模式中。

- Messaging（消息传递）模式 (129) [HoWo03] 用于组织这样的分布式软件系统：其不同的服务间通过交换消息来相互作用。有一系列相互连接的Message Channel和Message Router管理着跨网络的不同服务间的消息交换，包括传递请求和应答消息，消息中含有正常信息、元数据和错误信息。
- Publisher-Subscriber（发布者-订阅者）模式 (135) 用于组织这样的分布式软件系统：其

不同的服务和组件间通过一对多的关系异步地交换事件来相互作用。通常，事件的 Publisher 和 Subscriber 并不彼此知道对方。Subscriber 只负责消费事件，并不关心事件的 Publisher 是谁。类似地，Publisher 只负责供应事件，不关心谁订阅了这些事件。

- **Broker（经纪人）模式** (137) [POSA1] [VKZ04] 用于组织这样的分布式软件系统：其不同的组件间通过远程方法调用来相互作用。一个代理管理器管理着组件间互操作的各个关键方面，从派发请求到传递结果和异常等。

这些模式间有一些显著的区别，包括通信模型——比如多对一和一对多，以及应用组件间的耦合的程度。具体如下。

- 通过 **Broker** 模式，很多客户端可以调用托管在服务器上的远程组件对象的远程方法。因此客户端可以按照多对一的方式与服务器对象通信，并且它们描述功能的接口通常都是静态类型的。如果系统希望隐藏网络的存在，那么 **Broker** 提供的远程方法调用形式的通信就是最适合的选择。
- **Messaging** 模式降低了耦合度，放宽了对类型的限制。客户端动态地把固定类型的消息发送给特定的远程服务（它存在于通信的终端系统中），而不（必须地）发给特定的方法。因此 **Messaging** 模式可以实现多对一的通信方式，同时不需要静态地去预定义接口，否则这个接口会导致客户端对服务的依赖。
- **Publisher-Subscriber** 模式可以进一步解除应用组件之间的耦合。**Publisher** 和 **Subscriber** 以一对多的方式交换事件，而不必明确知道对方的身份，也不必在每次出现新的事件时都发出一个请求。因此，**Publisher-Subscriber** 模式的中间件负责追踪哪个 **Subscriber** 接收到了由 **Publisher** 异步发送的特定事件。**Subscriber** 接收到一个事件后，会执行一些操作，但是 **Publisher** 并不会直接启动 **Subscriber** 的某个方法的执行。

表 10-1 总结了上述区别。

表 10-1

模 式	通信方式	通信关系	组件依赖性
Broker	远程方法调用	一对一	组件接口
Messaging	消息	多对一	通信终端的消息格式
Publisher-Subscriber	事件	一对多	事件格式

在分布式系统中，想要实现完全的位置透明通信是不现实的[WWWK96]，但是 **Broker** 模式的确使得本地对象对远程组件对象的调用，无论是看上去还是执行效果都尽可能地像是在调用一个处于相同地址空间中的组件对象。如果有多个独立开发的自我完备的服务或者应用，它们需要协同工作并且组合为一个统一的软件系统，那么 **Messaging** 和 **Publisher-Subscriber** 模式最适合整合这种场景下的软件组件。**Messaging** 模式还允许不同的服务之间交换请求和响应。然而，**Publisher-Subscriber** 则是通过通知 **Subscriber** 状态的改变以及相关事件，来实现各个组件的整体协作的。

现实中，中间件产品和平台都会实现一种或多种模式。例如，**Web Services** 通过 **WS-NOTIFICATION**

实现了Publisher-Subscriber模式，通过SOAP实现了Messaging模式。CORBA通过通知服务（Notification Service）实现了Publisher-Subscriber模式，通过自身的ORB实现了Broker模式。有些CORBABroker的实现，比如BEA的Web Logic Enterprise，甚至实现于Tuxedo消息中间件之上。通常，可以把CORBA ORB Core看作是CORBABroker架构中的消息层。

有一些分布式计算的文献也提出过其他的通信方式，比如共享数据库、流化数据（data streaming）、文件传输和点对点传输[Fow03a][HoWo03][VKZ04]。我们认为这些应该算作对应用服务的某种精巧的组织方式，而不是用于交换信息的方式。正是由于存在这些不同，我们才把Pipes and Filters (116) 和Shared Repository (117) 放到了第9章而不是本章。流化数据方式可以通过Pipes and Filters模式来实现，此时的Pipes是基于Messaging或者Broker的中间件。类似地，共享仓库方式可以通过Shared Repository模式实现，在这个实现中可以使用基于Broker或者Publisher-Subscriber模式的中间件。

Messaging、Broker和Publisher-Subscriber是本章中其他模式的入口点。如果要想让中间件在分布式应用中发挥作用，那么每种中间件都必须能够解决很多不同的问题，并以此为基础，提供一套连贯的解决方案。因此将这些问题及其解决方案都以独立的模式进行文档化，就变得非常自然了。事实上，今天很多中间件平台的架构都是以这样的一些模式作为指导并记录下来的[SC99][ACM01][VSW02][VKZ04][MS03]。

基于Messaging模式的中间件可以细分为下面4种分布式基础架构模式。

- Message Channel（信道）模式 (130) [HoWo03]可以连接不同的应用服务，这些服务之间通过交换消息进行交互。一个服务将信息写入到通道中，另一个服务从通道中读取信息。
- Message Router（消息路由）模式 (134) [HoWo03]要求一个客户端根据不同的条件集，向应用的其他服务发送消息。
- Message Translator（消息转换器）模式 (133) [HoWo03]支持将消息转换为另一种形式。如果消息的发送者与读取者对消息的格式有不同的预期的话，可以使用该模式。
- Message Endpoint（消息端点）模式 (132) [HoWo03]可以通过封装和实现必要的适配代码，从而帮助应用服务与消息基础设施进行连接。

消息中间件是由Message Channel、Message Router、Message Translator和Message Endpoint模式以外更多的模式定义的。而这4种模式都分别引用了其他细粒度的模式，可以进一步分解和实现。我们没有详细地介绍这些细粒度的模式，而只是指出了它们最初的来源是*Enterprise Integration Patterns*[HoWo03]。正是所有这些模式，构成了我们分布式计算的模式语言的主要部分。

图10-1概括了Messaging模式和Publisher-Subscriber模式是如何与我们的模式语言整合在一起的。

Broker中间件的职责可以分解为下面的5种分布式基础设施模式。下面按照它们被应用在从客户端到服务器的顺序分别加以介绍。

- Client Proxy（客户端代理）模式 (139) [VKZ04] 为客户端提供了一个本地接口，用于与远程组件进行交互。客户端能够以一种与位置无关的方式访问远程组件，使得远程组件看起来是与本地组件在一起的。

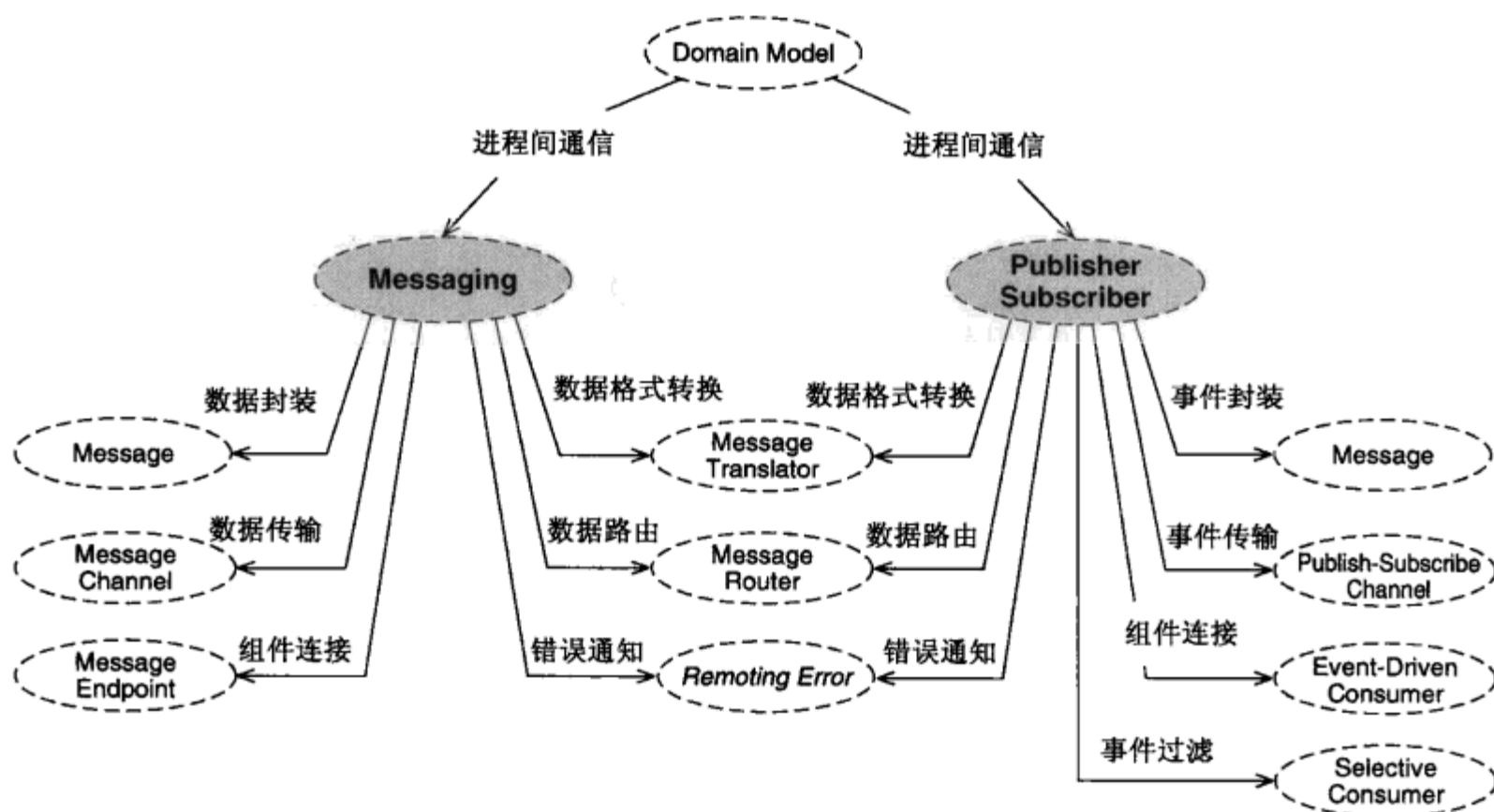


图 10-1

- Requestor（请求者）模式 (140) [VKZ04] 封装了客户端远程通信的各种细节，比如封装（marshaling）、通过网络发送请求、允许客户以位置无关的方式访问远程组件。
- Client Request Handler（客户端请求处理程序）模式 (143) [VKZ04] 封装了客户端在统一的接口背后进行的进程间通信的细节。
- Server Request Handler（服务器请求处理程序）模式 (144) [VKZ04] 封装了服务器端在统一的接口背后进行的进程间通信的细节。
- Invoker（调用者）模式 (142) [VKZ04] 可以在服务器组件收到来自远程客户端的请求时，替服务器组件屏蔽掉与网络相关的问题，比如请求的接收、解封送（demarshaling）、分派请求。
- Client Proxy、Requestor、Client Request Handler、Server Request Handler、Invoker 这些模式自身也会通过其他一些分布式基础设施模式进行细化。我们同样不会详细地讲述其他那些模式，而只是给出它们最初的来源是 *Remoting Patterns* [VKZ04]。

熟悉 POSA 系列第 1 卷（模式系统）的读者可能注意到了，Client-Dispatcher-Server 和 Forwarder-Receiver 这两个模式 [POSA1] 没有出现在上面的模式列表中。由于这两个模式的职责已经包含在模式语言的其他模式中了，所以我们省略了它们。根据我们的经验来看，这些模式覆盖的范围过于宽泛，建议将它们重构到多个更小的、更有针对性的模式中。Client-Dispatcher-Server 模式的职责可以通过 Broker 模式实现，在 Broker 中使用 Lookup(292) 服务；Client Request Handler (143) 和 Server Request Handler 相关联，则形成一个 Forwarder-Receiver 结构。

图 10-2 表现了在我们的模式语言中，Broker 模式是如何与其他模式一起协作进行分布式计算的。

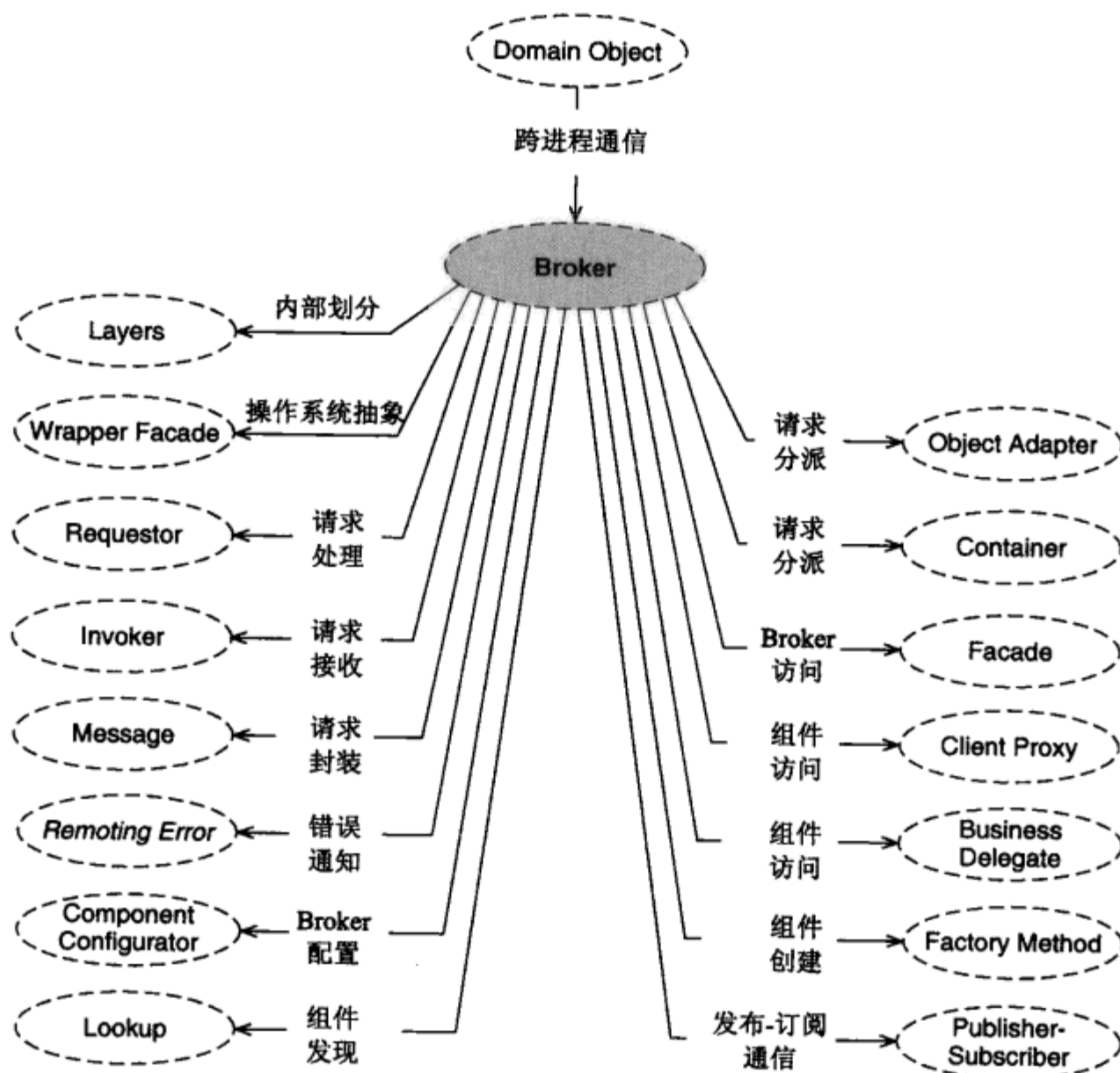


图 10-2

图 10-2 以及图 10-1 显示了 Publisher-Subscriber 与 Messaging 及 Broker 之间的关系。Publisher-Subscriber 的匿名、异步组通信模型可以使用 Messaging 或者 Broker 实现，因此它可以视为后两种模式的特殊形式。我们将它作为一个独立的模式分离出来，是因为它解决的问题的关注点与 Messaging 和 Broker 有所不同。最终促使分布式应用组件之间的通信耦合更松，而且伸缩性更好。将 Publisher-Subscriber 描述为一个独立的模式，而不是其他模式的特例，也可以让我们更加明确地讨论其独特的关注点和效果。

我们必须承认，这些分布式基础设施模式并不能为分布式系统带来彻底的位置透明的通信 [WWWK96]。虽然基于 Broker、Messaging 和 Publisher-Subscriber 的中间件平台可以为应用肩负起很多繁琐的、易出错的网络编程任务，但归根结底，它们只是分布式应用组件之间的连接器。那些组件还要处理好它们自身需要面对的真正挑战。

例如，一个分布式应用必须在遇到不可避免的网络失败或者服务器宕机的情况下，可以从中恢复。同样的，一个调用了远程组件上服务的客户端，必须考虑网络为远程通信带来的延迟和抖动。另外，单独依靠分布式基础设施无法解决组件间的问题，比如跨网络的各个组件是否正确部

署或者能否协调运作。分布式组件的特定部署，以及这些组件处理网络失败、延迟、抖动的方式，也就成了影响系统可伸缩性、性能和稳定性的重要因素[DBOSG05]。

换句话说，基于Messaging、Broker、Publisher-Subscriber的中间件平台是分布式系统中重要的组成部分，然而它们不能处理特定于应用的问题，这超出了它们的职责范围。分布式系统的组件必须仔细地定义——时刻要记得网络的特点——尽管Messaging、Broker、Publisher-Subscriber中间件允许一个组件可以与其他组件的特定位置无关。

10.1 Messaging**

当在多个处理程序或者网络节点上部署Domain Model (106) 或者Pipes and Filters (116) 时……我们经常需要一个通信基础架构，将独立开发的服务整合到一个完整的系统中。



有些分布式系统是由独立开发的服务组成的。但是，倘若要将它们组成完整的系统，这些服务必须能够可靠地交互，同时，每个服务之间又不能产生过分紧密的依赖关系。

要将现有的、自我完备的、用于特殊目的服务组合成一套（企业级）解决方案，应用整合是其中一项关键技术。每个独立的服务都提供了各自的业务逻辑和业务价值，但是要把它们整合在一起后方能提供一个完成的企业业务流程和价值链。欲将这些独立的服务整合到完整的应用中，很自然地需要这些服务彼此可靠地协作。但是，由于这些服务都是独立开发的，因此它们通常都不知道别的组件有哪些特定的功能接口。另外，每个服务都可能在多种背景下参与整合，所以在一定的背景中使用，不应该排除在其他背景中使用的可能性。

因此，通过消息总线连接不同服务，允许它们异步地传递数据消息。对消息进行编码，这样发送者和接收者不必静态地了解所有的数据类型信息即可可靠地通信（见图10-3）。

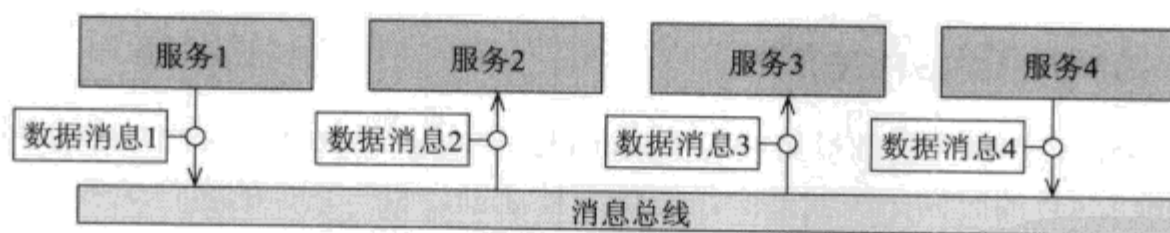


图 10-3

构成分布式系统的服务通过消息总线连接在一起，利用消息总线与其他服务交换数据消息。客户端可以向远程服务发送一个异步的消息，从而开始一次协作。远程服务会处理收到的消息，如果需要返回值的话，远程服务会通过包含处理结果的消息，异步地把响应返回给客户端。消息通常是自描述的——它们既包含描述Messaging结构（schema）的元数据，也包含结构中相应的数值。



基于Messaging的中间件方便了分布式应用中各个服务的交互，它们不必自己处理有关远程的问题，不必依赖于静态定义的服务接口和数据结构。另外，Messaging通信的异步特质允许分

布式应用的服务同时处理多个请求，而不需要阻塞；同时也能够参与到多个应用的整合与使用背景中。Messaging带来了松耦合，这是企业应用整合(Enterprise Application Integration, EAI)[Lin03]与面向服务架构(Service-Oriented Architectures, SOA)[Kaye03]的关键。Messaging的主要缺点在于它缺少静态定义的接口，这导致我们很难在运行之前校验系统的行为，另外，处理自描述的消息，也潜在地存在着很高的时间与空间开销[Bell06]。

用于在应用服务之间交换的数据通常被封装在Message(245)中。一个消息对它的发送者与接收者以及Messaging中间件自身，都隐藏了其内容的具体数据格式，这样对数据格式的修改对于系统就是透明的了。XML是用于表现自描述消息的元数据和数值的流行的格式。它使得客户端可以生成并发送消息，这些消息的形式和内容不需要静态地确定下来。

Messaging客户端只知道它们所用的服务端点(endpoint)，并不知道服务的特定接口。因此，客户端在将消息发送到特定的服务之前，无法静态地检查消息的格式和内容。最终，理解消息的格式和内容的责任落在了接收消息的服务上。这个过程通常会涉及消息的动态解析，包括验证和提取消息的内容。如果服务无法理解消息中的一些域，可以直接将它们忽略，因为有些服务的消息格式最初设计的时候并没有考虑它们的协作，这种方式简化了它们之间的整合。

一个具体的Messaging通常包含若干个特殊的部分。Message Channel(130)支持彼此交互的远程服务之间的点对点通信，也增加了消息交换的可靠性。Message Endpoint(132)利用Messaging中间件连接不同的应用服务，这些中间件不必依赖于具体的消息API，就可以发送和接收消息。这样，底层的Messaging基础设施可以透明地进行替换和升级。

如果消息的发送者和接收者之间不能共享一个公共的消息格式，Message Translator(133)可以将发送者产生的消息转换为接收者可以理解的格式。如果发送者不知道将消息发往何处，Message Router(134)可以帮助它定位到预期的接收者上。如果Messaging中间件在内部不能处理一个通信失败，则可以将它作为一个Remoting Error [VKZ04] 返回给发送它的客户端。

10.2 Message Channel**

在开发Messaging(129)基础设施或者Broker(137)中的Client Request Handler(143)和Server Request Handler(144)的时候……我们必须提供一种途径，将一系列的客户端和服务连接起来，它们会通过发送和接收消息进行通信。



在分布式系统中，基于消息的通信方式提供了不同服务间的松耦合。消息中只包含了用于在不同的客户端和服务之间进行交换的数据，所以消息自身并不知道谁真正需要它们。

松耦合可以降低整合各种信息系统时的难度，但是，每个松散的端点必须以某种方式联系起来。一个客户端随机地发送消息，同时服务不假甄别，随机地接收消息，通常这难以满足系统的要求。发送消息的客户端知道这些消息中包含了何种类型的信息，通常它们也知道消息预期的接收者。类似地，接收消息的服务会寻找它们能够处理的特定消息，通常都是来自于特定的发送者。换句话说，客户端和服务之间需要以一种可预见的、可靠的方式交换消息。

因此，使用Message Channel将彼此协作的客户端和服务连接起来，使它们可以交换消息(见

图10-4)。

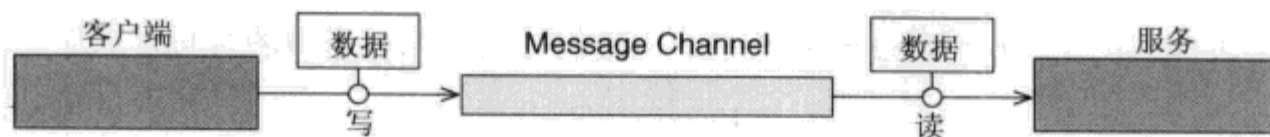


图 10-4

当一个客户端有消息准备通信时，它将这个消息写到Message Channel中。处理它的服务能够从Message Channel中提取它并进行处理。



Message Channel可以连接多个交互的客户端和服务，因此这些客户端和服务能够以良好定义和可靠的方式交换消息。负责将消息写入通道的客户端确信从通道中读取消息的服务一定会对其所包含的信息感兴趣，同时读取消息的服务也可以确定接收到的消息是它们可以使用和处理的。

因此，对于服务和客户端来说，Message Channel是用来读写消息的一个非常合理的途径。Message Channel常见的类型包括：Point-TO-Point Channel [HoWo03]可以严格地连接一个客户端与一个服务，确保只有它们才能读取写入其中的消息；Publish-Subscribe Channel [HoWo03]保证Publisher会利用Publisher-Subscriber模式(135)，将消息广播给多个Subscriber。

利用Invalid Message Channel [HoWo03]，可以解耦错误消息的处理，将其逻辑与其他的应用逻辑分离。另外，对于被成功发送却无法被转发的消息，可以用Dead Letter Channel [HoWo03]来处理。最后，Datatype Channel [HoWo03]可以确保在一个通道中所有的消息都是同一类型的，这样就降低了在消息预期的接收方进行消息校验的开销。

一个Message Channel至少会被两个并发的实体所共享：向通道中发送消息的客户端，以及从通道中获得消息的服务。因此，根据通道的实现和使用，它可能需要同步。Thread-Safe Interface (224)可以实现通道的接口上的同步。如果实现为Monitor Object (214)，则能够支持同时访问通道时的并发控制。

通常，哪种Message Channel配置是最合适的，是由分布式应用运营性需求决定的。例如，如果要求保障信息的安全性，可能要对安全敏感的协作使用单独的Secure Channels [SFHBS06]。如果要求性能和可伸缩性，可能同样要对每一种类型的消息（甚至是每一个用例）使用独立的Message Channel。

但是，Message Channel并不是免费的午餐，因为它需要内存、网络资源、持久化存储来维持Guaranteed Delivery [HoWo03]。因此，开发者必须明确地、清晰地计划并配置Message Channel的数量和类型，以确保在部署系统时，服务可以达到预期的质量。一个设计良好的Message Channel集合，可以形成一个Message Bus [HoWo03]，它在分布式系统中扮演了客户端和服务之间类似于消息API的角色。

在设计客户端和服务时，如果并未打算使用Message Channel或者Message Bus，可以通过一个Channel Adapter [HoWo03]将其连接到通道或者总线上面。如果客户端与服务需要使用不同的通道或者总线的实现时，可以采用Messaging Bridge [HoWo03]来把它们连接起来。

10.3 Message Endpoint**

在开发Messaging (129)基础设施的时候……我们必须确保应用中的客户端和服务能够发送和接受消息。



通常，在独立的应用中，客户端和服务靠彼此传递数据来完成协作。但是，当客户端和服务是由一个消息基础设施连接起来的时候，这种直接的协作就变得不可能了：我们需要在数据和消息之间进行双向转换。

在应用内部直接执行数据-消息之间的转换，将会导致在应用与（消息中间件所需的）特定的消息格式之间形成紧密的耦合。因此，如果这样的话，将难以在不同的应用中使用相同的服务。而且，在基础设施代码中混入了特定领域的代码后，将增加修改与维护的难度。即使把消息机制放入应用的基础部分，要想替换底层的消息基础设施，仍然非常耗时、冗长，并且容易出错。

因此，使用特定的Message Endpoint将应用中的客户端和服务同消息基础设施连接起来。这样客户端就可以和服务交换消息了（见图10-5）。

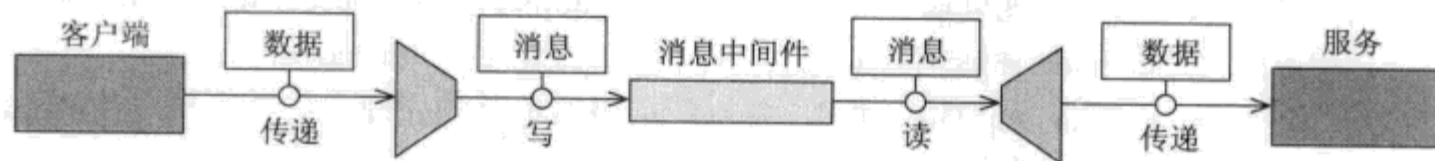


图 10-5

当客户端中出现数据需要通信时，它会把数据传递给其相关的Message Endpoint，后者首先会把数据转换为消息中间件能够理解的消息，然后将消息发送到另一个Message Endpoint，该端点代表了消息的接收方。这个端点会把消息转换为能被接收服务所理解的数据，然后以正确的格式将数据传送给该服务。



Message Endpoints为应用的客户端和服务封装了消息中间件的细节，并且为它们对中间件的通用API做了专门的客户化。消息API的变更，甚至整个消息基础设施的改变，都可以对应用完全透明。另外，所有必要的修改都局限在端点内。

通常，Message Endpoint应该设计为Messaging Gateway [HoWo03]，以便封装以消息为中心的代码，并且向它所代表的服务公布一个以领域为中心的接口。在Message Endpoint的内部可以部署一个Messaging Mapper [HoWo03]，它能够在服务与消息之间传递数据。为了提供对同步方法的异步访问能力，可以将Message Endpoint组织为一个Service Activator [HoWo03]。Transactional Client [HoWo03]允许Message Endpoint在消息中间件中显式地控制事务。

Message Endpoint可以在众多不同的接收消息的方法中做出选择。Polling Consumer [HoWo03]提供了一种前摄的（proactive）消息接收策略，它只在所代表的服务准备接收消息的时刻才会读取它们。相反，一个Event-Driven Consumer [HoWo03]支持反应式的（reactive）消息接收策略，它会在消息到达时立刻着手处理。如果服务实现了无状态的功能，则Message endpoint可以实现

为Competing Consumer [HoWo03], 以便允许多个服务的实例可以并发地处理消息。如果多个服务共享同一个Message Endpoint, 则Message Dispatcher [HoWo03]可以把外部传来的消息分派给“正确的”接收者。

将Message Endpoint设计为Selective Consumer [HoWo03], 就可以对外来的消息进行过滤, 服务只处理符合过滤器规则的消息。Message Endpoint也可以是一个Durable Subscriber [HoWo03], 这样, 在服务不可用的时候, 接收到的消息也不会丢失。最后, Message Endpoint也可以实现为Idempotent Receiver [HoWo03], 它能够处理在意外情况下多次收到的消息。

Message Endpoint所代表服务类型和服务功能决定了上述消息接收策略中, 哪一个对某个特定Message Endpoint是最适合的。Message Endpoint的开发可以针对“它的”服务进行定制。

10.4 Message Translator**

在开发Messaging (129) 基础设施的时候……通常, 我们必须将消息从客户端发布的格式转换为接收消息的服务所能理解的格式。



使用消息, 可以使得应用的客户端与服务之间以松耦合的方式进行通信。但是, 经过这样的解耦以后, 发送消息的客户端不能假定接收消息的服务能够理解相同的消息格式。

在有些复杂的整合场景中, 现存的和独立开发的组件将一起组成新的应用, 这时很多服务都将使用特定的消息格式。如果要在服务内部解决各种“语言”带来的“巴别塔”式的混乱, 服务之间将会引入显式的、彼此交错的依赖关系, 这不仅与松耦合的思想相悖, 而且会减少基于消息通信所带来的好处。另外, 统一所有服务的消息的想法也是不切实际的, 因为这会削弱它们在其他应用或者整合场景中的可用性。

因此, 在应用的客户端与服务之间引入Message Translator, 它可以将消息从一种格式转换为另一种格式 (见图10-6)。

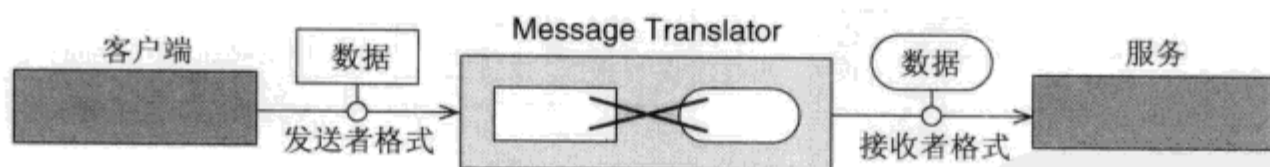


图 10-6

Message Translator提供了双向的消息格式转换。在特定的协作场景中, 客户端可以使用任何格式来发送消息。Message Translator能够保证服务接收到的消息的格式是它们所能够理解的。



即使客户端和服务不能共享通用的消息格式, Message Translator仍然能够将Messaging引入松耦合风格的通信行为保留下来。另外, 所有用于转换消息的代码都局限在一个专门的模块中。这种设计使得格式的变化对于通过Message Translator交换消息的客户端与服务来说是独立的、透明的。

在很多整合场景中, 通过在消息头的格式和内容上设置特定的需求来支持消息交换。Envelope Wrapper [HoWo03]可以封装消息的载体, 这样它可以遵循消息基础设施要求的格式。当

消息到达目的地后,消息的载体可以解除封装。在消息中,如果原始的客户端无法满足目标服务需要的数据域,这时就需要使用Content Enricher [HoWo03],它能够通过现有可用的数据查找或者计算出缺失的信息。Content Filter [HoWo03]可以完成相反的工作:从消息中移除不必要的数。Claim Check [HoWo03]与Content Filter类似,不过它能够保存移除的数据,以便在后期重新获取。Normalizer [HoWo03]可以将多种不同的消息格式转换为一种通用的格式,而一个独立于任何特定服务的Canonical Data Format [HoWo03]可以在任何消息中间件内部使用,从而将应用内的消息转换工作减少到最低。

10.5 Message Router**

在开发Messaging (129)基础设施的时候……我们必须选择一条路由途径,将消息从发送端传递给接收端。



在相互协作的客户端和服务之间交换的消息,必须通过消息基础设施进行路由。但是,这三者都不应该知道如何选择路由途径。

让应用的客户端和服务自行决定消息在系统中传递的路径,这并不是解决路由问题的有效方案,而且当它们收到与自己无关的消息时,也不应该由它们负责转发这些消息。这种设计会导致应用的代码与基础设施的代码之间存在紧密的耦合,客户端与服务会依赖于消息基础设施的内部结构和配置。因此,当发生变化时,会引入维护的问题。类似地,让消息负责自身的路由,也会为协同工作的组件间交换的数据带来相同的问题。

因此,提供Message Router,它能够根据一系列的条件,接收来自Message Channel的消息,然后把它们重新插入到不同的Message Channel中(见图10-7)。

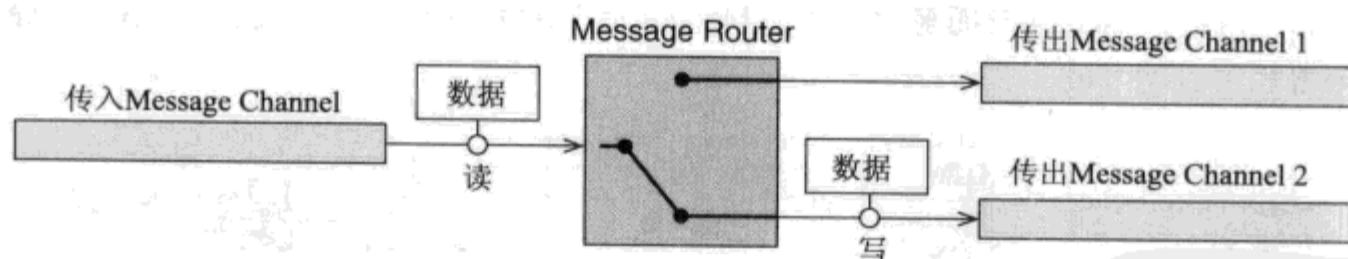


图 10-7

Message Router可以将一系列的Message Channel连接成一个网络。它从一个通道中读出消息,将它路由到另一个通道里,后者可以将消息传递给它们预期的接收者。



Message Router最大的好处在于它可以把有关消息将发送给哪个目标的规则集中到某个特定的位置,并将这种决策规则与应用的客户端、服务以及它们之间交换的数据隔离开。如果定义了新的消息类型,在Message Router内部的路由规则可以很容易地做局部的修改。如果必要的话,可以在消息中间件中插入新的Message Router。因此,当开发者需要在客户端与服务之间发送消息时,就拥有了更多的选择。同时,开发者还可以独立地改变路由规则,这些对应用来说是透明的。

Message Router的缺点在于它给应用增加了额外的处理过程，这可能降低应用的性能。Message Router还必须知道与它相连的所有Message Channel。如果配置的变化很频繁，这可能会带来很高的维护成本。而且，如果没有相关工具的帮助，系统中包含的Message Router越多，我们就越难以分析和理解消息在系统中经过的完整路线。

所以，在部署系统的时候，开发者必须仔细地计划和配置Message Router的数量和类型，以满足服务需要达到的质量要求。系统配置中包含的Message Router越多，消息在应用的不同组件之间进行路由的方式就越灵活，但是消息交换的效率也会越差。因此，通常我们总要选择满足应用需求的Message Router的最小集合，这样才能在应用需求和简单性、灵活性，以及服务质量之间取得平衡。

Message Router的种类有很多。Context-Based Router[HoWo03]的路由判断是基于环境条件的，比如系统负载、故障转移场景(failover scenario)，或者是否需要系统监视Detour(system monitoring Detour)[HoWo03]。相反，Content-Based Router[HoWo03]在决定消息的目的地时，会使用消息的某些特定属性，比如它们的类型或者内容。Message Filter[HoWo03]可以协助Content-Based Router丢弃掉与路由规则不匹配的消息。Recipient List[HoWo03]可以通过它收到的消息，确定出一个消息接收方的列表。Process Manager[HoWo03]根据它之前路由的消息的中间结果来路由后面的消息。Message Broker[HoWo03]提供了中央辐射架构，在整个应用中路由各个消息。

其他的Message Router有助于管理在不同组件间交换的消息。Splitter[HoWo03]可以把一个大型的消息转换为多个小型的消息，这样可以分别独立地路由。Aggregator[HoWo03]提供了相反的功能：将多个消息合并为一个消息。Resequencer[HoWo03]可以收集失序的消息并重新进行排序，以便这些消息可以以正确的顺序重新发布。Routing Slip[HoWo03]可以在消息发送到接收者之前，显式地为消息增加路由信息。Composed Message Processor[HoWo03]能够利用Splitter将一个消息分解为多个部分，再针对每一部分的消息执行一些处理，然后通过Aggregator将各个部分重新组装到一起，再把它定向到一个输出通道上。最后，Scatter-Gather[HoWo03]可以将一个消息广播到多个接收者，然后利用每个接收者的反馈消息，创建一个统一的聚集响应消息。

通常，实现Message Router的控制逻辑有两种选择：要么是静态配置的，要么是动态配置的。静态配置的Message Router的运行时开销很低，但是缺少灵活性；动态配置的Message Router则完全相反。我们可以在中央Control Bus[HoWo03]的协助下实现动态配置，或者将路由实现为一个Dynamic Router[HoWo03]，它可以基于来自于潜在的消息接收者的信息来配置自身。

从多个输入通道接收消息的Message Router必须是同步的。Thread-Safe Interface(224)可以强制地在路由的接口上执行同步，另外实现为Monitor Object(214)可以支持全部Message Channel的协作并发控制，并且允许路由并行地接收消息。

10.6 Publisher-Subscriber**

在把Domain Model(106)部署到多个处理程序或者多个网络节点上的时候……我们通常需要一种基础设施，以支持应用的各个组件把彼此关注的事件通知给其他组件。



在一些分布式应用中，组件是松耦合的，并且可以独立地运行。然而，如果这种应用需要将某些信息传递给部分或者全部其他的组件，那么就需要一种通知机制，以告知组件相关状态的变化和其他需要关注的事件，这些事件会影响到或者调整组件自身的计算。

无论如何，这种通知机制不应该在应用的组件间产生过度的耦合，否则组件将失去它们的独立性。应用组件只希望知道系统中有一个组件正处于某种特定的状态中，而不关心具体是哪一个组件。类似地，发布事件的组件通常都不关心哪些组件希望收到这些信息。另外，一个组件不应该依赖于如何找到其他组件，或者依赖于其他组件在系统中的特定位置。

因此，定义一个用于传播变更的基础设施，允许在分布式应用中的Publisher可以发送事件，事件可以承载对其他组件有用的信息。这些信息发布后，要通知对这些信息感兴趣的组件（见图10-8）。

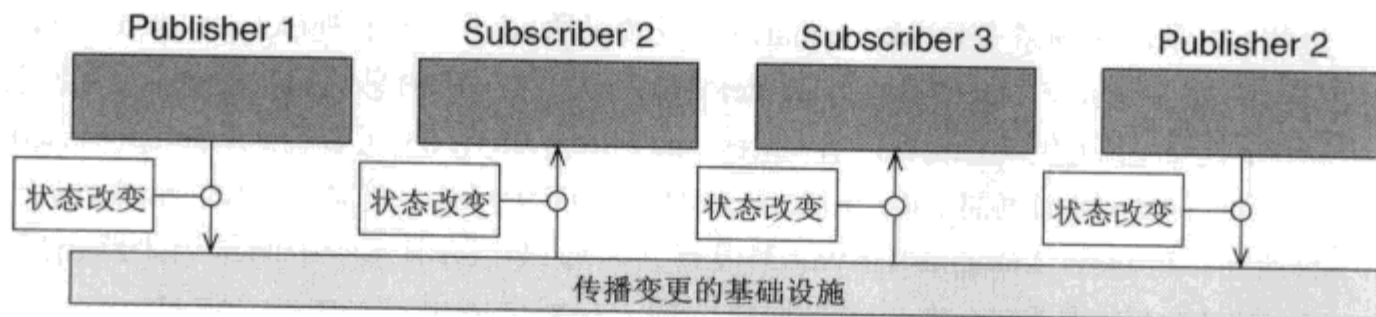


图 10-8

Publisher和Subscriber都会被注册到用于传播变更的基础设施上，并通知Publisher可以发布的事件类型，以及Subscriber可以接收的事件类型。基础设施利用这些注册信息，将事件从Publisher通过网络传递给对该事件感兴趣的Subscriber。Subscriber从基础设施接收到事件后，可以使用事件中的信息来指导或者协调它们自身的计算。



与Messaging一样，Publisher-Subscriber也支持异步通信，也就是说，Publisher向Subscriber传递事件后，不需要为等待后者的响应而阻塞。异步性有助于Publisher与Subscriber之间的解耦，这样它们可以在不同的时间点激活和启用，这正是分布式系统中固有的并行特性。另外，Publisher-Subscriber允许应用中的组件可以异步地协调计算，不必带来与另一个组件间显式的依赖：它们不知道也不依赖于另一个组件的位置和标识，因为它们只发送和接收状态的变化和/或变化的状态。

只有基础设施才知道组件的连接方式和位置，以及事件在系统中路由的方式。Publisher-Subscriber还支持组通信，这是指事件的Publisher不需要显式地通知每一个Subscriber，而基础设施会将事件转发给所有关注它的Subscriber。

这种匿名通信的缺点在于，如果Subscriber只对某一特定类型的事件感兴趣，并且仅当事件的内容符合特定的规则时Subscriber才会有所响应，那么匿名通信将带来不必要的开销。解决这个问题的方法之一是基于事件的类型或者内容进行过滤。然而，过滤又会引发额外的开销。比如，如果在Publisher-Subscriber中间件内部进行过滤，就会降低系统的吞吐量；如果在Subscriber内部进行过滤，就会有一些不必要的通知；如果在Publisher内部进行过滤，则会破坏匿名通信的模型。

Publisher-Subscriber中间件会将多个组件连接起来。用于在这些组件之间交换的信息被封装在事件内部。这些事件被实现为Message (245)。事件可以对Publisher和（若干）Subscriber以及Publisher-Subscriber中间件隐藏具体的消息格式，这就使得消息格式的变化对于系统是透明的了。

Publisher-Subscriber中间件可以通过不同的方式实现。一种方法涉及Messaging和Broker中间件的重用。例如，可以利用现有的Messaging和Broker中间件实现Publisher-Subscriber中间件，现实中，这两种中间件产品分别对应于很多WS-NOTIFICATION [OASIS06c] [OASIS06c]和CORBA Notification Service [OMG04c]。另一种方法是使用基本的并发和网络编程模式[POSA2]实现Publisher-Subscriber，比如很多DDS [OMG05b]产品。总体上说，第一种方法可以简化中间件开发者的工作，而第二种方法会带来更好的性能。

为了支持匿名的、异步的组通信，可以利用多个Publish-Subscribe Channels [HoWo03]或者Event Channels [HV99]，采用广播或者多址传播的方式，将事件消息从Publisher传送给Subscriber。组件把自身会将发布哪些事件以及将接收哪些事件的信息，通知给特定的通道。Event-Driven Consumers [HoWo03]支持以透明的方式将消费者适配到特定的通知publish/subscribe API上。这种设计，使得底层的Publisher-Subscriber基础设施的改变或更新对于系统其他部分是透明的。将Subscriber实现为Selective Consumers [HoWo03]，可以对接收到的事件消息进行过滤——Subscriber仅处理内容符合过滤条件的事件。

如果处理同一事件的Publisher和Subscriber不能共用一个统一的消息格式，Message Translator (133)可以将Publisher生成的事件转换为可以被Subscriber所理解的格式。Message Router (134)可以帮助将事件通过中间件路由到注册的Subscriber上。如果通信过程中产生的失败不能在Publisher-Subscriber中间件内部处理，则可以将它作为一个Remoting Error [VKZ04]返回给发送该事件的Publisher。

10.7 Broker**

在把Domain Model (106)部署到多个处理程序或者多个网络节点上的时候……我们通常需要一个通信基础设施，它可以对应用屏蔽掉组件的位置和IPC的各种复杂性。



分布式系统总会面对很多在单进程的系统中不会遇到的挑战。然而，应用的代码不应该直接应对这些挑战，而是应该使用模块化的编程模型来简化应用的开发，使应用可以与网络 and 定位的细节隔离开。

在分布式系统中，向服务发送请求是很困难的。这种复杂性的来源之一就是需要找到各种服务，它们可能是由不同的语言所编写，并且运行在不同的操作系统上的。如果服务和某个特定的上下文耦合得过于紧密，那么在将它们迁移到另一个分布式环境中或者在其他分布式应用中重用它们时，将会既耗时又耗力。复杂性的另一个来源是：如何确定服务实现在分布式系统中的部署位置和部署方式。在理想的情况下，服务之间的交互应该是以一种通用的、位置无关的方式调用另一个服务的方法，而不必关心对方是本地的还是远程的。

因此，在分布式系统中，通过使用一系列的Broker，可以从应用的功能中，隔离并封装通信

基础设施的细节。定义一个基于组件的编程模式，这样客户端可以像使用本地服务一样，调用远程服务的方法（见图10-9）。

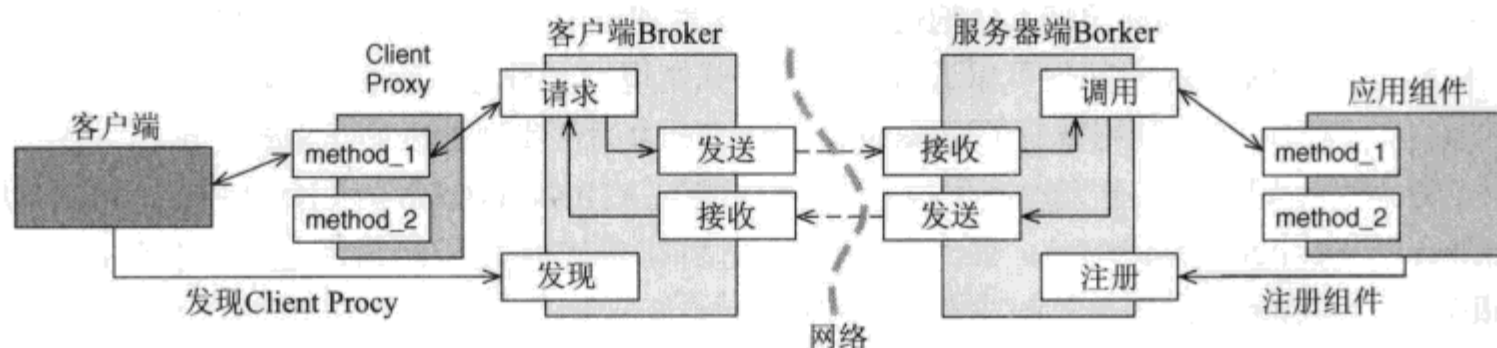


图 10-9

对于每一个参与系统的网络节点，我们至少要为它定义一个Broker实例。组件的接口和位置被注册到它们本地的Broker上，以此来获得在分布式系统内部的可见性。为了调用组件上的函数，客户端会向本地的Broker请求一个Proxy（代理），它扮演了远程组件的代理人角色。客户端会调用Proxy的方法以发起到远程组件的请求。Proxy通过与客户端和服务端的Broker协作，来向组件发送请求，接收任何响应结果。



Broker使得分布式应用中的组件在交互的时候不需要处理与远程相关的问题。它也可以根据客户端和服务端组件的位置选择使用远程方法调用还是同站方法调用^①（collocated method calls），这可以使得我们的通信机制更为完善。例如，如果客户端和组件在同一个地址空间内，则Broker可以优化通信路径，以减少不必要的开销。

大多数Broker实现都依赖于Layers (108) 架构去管理复杂度，比如CORBA [OMG04a]和Microsoft .NET Remoting [Ram02]。这些Layer可以进一步分解到“特定目的”的组件中，来完成特定的网络和通信任务。我们以CORBA中的分层方案（layering scheme）[SC99]为例，来说明这种划分方法——其他的分层方案和中间件可能会涉及不同的分配方式 [VKZ04]。

一个OS适配层可以向Broker隐藏其底层的执行平台。在使用虚拟机的语言中，比如Java，这个层就是虚拟机。在其他的语言中，比如C++，适配层通常包含了一系列的Wrapper Facade (269)，为特定的OS API提供一套统一的接口。

一个ORB核心层构成了Broker设计的核心。通常它是一个包含了两个组件的消息基础设施（messaging infrastructure）。一个是Requestor (140)，它可以将请求Message (245)从客户端转发到被调用的远程组件的本地Broker；另一个是Invoker (142)，它负责接收由客户端Broker发送的请求消息，并将这些请求分派给相应的远程组件。如果通信故障不能在的Broker基础设施内部解决，我们可以向发出该请求的客户端报告一个Remoting Error [VKZ04]。Component Configurator (289)可以使用特定的通信策略和协议来配置Requestor和Invoker的实现。这样我们就可以支持Broker内部透明的替换和更新了，同时，我们还能在协议层次上将异构的或者遗留的组件整合到分布式

① 表示运行于客户端线程内的方法调用。——译者注

系统中。

额外的Lookup (292) 功能允许组件向Broker基础设施注册自己的接口和位置。客户端可以使用Lookup查找和访问这些组件。使用Lookup, 客户端不必知道组件的具体位置, 但又能在运行时连接到它们。Lookup也使组件的部署更加灵活, 这样既能够优化网络资源的使用, 又能应对不同的部署方案。为了完善远程过程调用, Broker可以与事件通道(event channel)交互, 以支持基于事件的通知机制, 这在本质上是一个Publisher-Subscriber (135) 服务。

典型的ORB适配器能够提供一个Container (288), 来管理远程组件的技术环境(technical environment)。ORB适配器与一系列的Skeleton^①交互, 后者实际上是Object Adapter (256), 负责Broker的通用消息基础设施和远程组件的特定接口之间的映射。Facade (171) 呈现了简单的接口, 组件可以通过它来访问本地的Broker。

Client Proxy (139) 表示客户地址空间中的组件。Proxy提供了统一的接口, 将组件上特定的方法调用映射到Broker面向消息的通信功能上。Proxy允许客户端像访问本地组件一样访问远程组件, 并可以用来透明地实现同站优化(collocation optimizations) [ScVi99]。

Client Proxy可以提高分布式系统中通信的位置无关性, 但是无法实现完全的透明性。客户端在使用Client Proxy之前, 必须先从其本地Broker中找到它(比如通过Lookup)。对于本地组件来说我们不需要这个动作, 除非我们要用Factory Method (313) 来访问所用到的组件。另外, Client Proxy可能无法对它们的客户端完全透明地处理所有的Remoting Errors。我们可以使用Business Delegate (170) 来封装这些“有关基础设施的问题”, 这有助于为分布式系统中的各个组件提供更彻底的位置无关的通信。

10.8 Client Proxy**

在构建客户端Broker (137) 基础设施, 或者为分布式组件实现基于Proxy (169) 的接口, 或者为远程组件实现Business Delegate (170) 的时候……我们必须提供一种抽象, 允许客户端使用远程方法调用来访问远程组件。



在访问远程组件服务的时候, 客户端需要使用特定的数据格式和网络协议。但是, 如果将格式与协议信息直接硬编码到客户端应用中, 将导致客户端依赖于协作组件的“远程性”, 因为远程组件的调用与本地组件的调用是不同的。

理想情况下, 对组件的访问应该是位置无关的。本地组件的方法调用和远程组件的方法调用之间不应该有任何功能性区别。

因此, 在客户端的地址空间中提供一个Client Proxy, 它可以代替远程组件。Proxy提供了与远程组件相同的接口, 并将客户端的调用映射到特定的消息格式和协议, 以便跨网络发送这些调用请求(见图10-10)。

^① 参考第6章第1节对Stub和Skeleton的介绍。——译者注

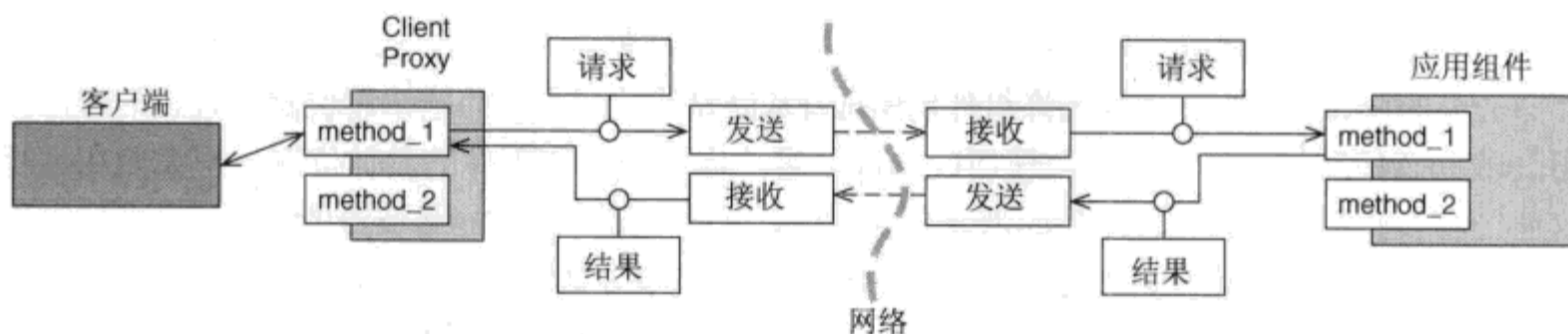


图 10-10

确保客户端应用的请求只能通过Client Proxy发送到远程组件。接下来，Proxy将会把具体的方法调用和参数转换为可以被网络理解的数据格式，然后使用IPC机制将数据发送到远程组件。Proxy所表示的那个组件返回处理结果，Client Proxy再将返回的结果转换为客户端所能理解的格式。



Client Proxy支持远程方法调用风格的IPC。这样调用本地组件和调用远程组件之间就没有API上的区别了，这提高了分布式应用内部通信的位置无关性。另外，Client Proxy可以为客户端屏蔽掉远程组件接口上的变化，这样可以避免在更新组件时产生涟漪效应。然而，要把Proxy的接口设计为网络不敏感的风格，可能带来额外的开销。例如，它可能需要提供很多细粒度的方法，比如为Proxy所代表的组件的每一个可见属性提供访问器（accessor）。客户端访问所有属性将需要很多次Proxy调用，每一个调用都会带来网络开销。

另外，Client Proxy只是有利于——而不能完全实现——通信的位置无关性。客户端在使用Client Proxy之前，必须先通过它们的本地Broker找到它——因此，客户端组件远程属性其实是知道的。而且Client Proxy可能无法对客户端透明地处理所有由网络返回的错误。

Client Proxy可以使用Resource Cache (299) 来维护它所代表的远程组件的不变（immutable）数据和状态，只需要数据和状态是第一次访问和传递即可。缓存机制可以避免不必要的性能开销，以及后续访问这些数据和状态的网络通信。如果把不变的状态和数据封装在Immutable Value (231) 中，Client Proxy可以将它直接传递给客户端。Client Proxy可以使用Authorization (204) 来确保客户端有权限访问远程组件。这样，如果访问被拒绝，可以降低不必要的性能开销和网络通信。

将Client Proxy设计为Remote Facade [Fow03a]有助于解决性能问题，Remote Facade会把相关的细粒度的方法合并到一个单独的粗粒度的方法中，比如为了响应一个单独的调用，这个粗粒度的方法可以返回远程组件的所有可见属性。

10.9 Requestor**

在构建客户端Broker (137) 基础设施的时候……我们必须提供一种方法，通过网络向远程组件发送方法调用请求。



客户端发起的对远程组件的方法调用涉及大量管理和基础设施方面的任务。在每一个客户端中重复地实现这些任务是非常繁琐的，而且容易出错，同时，应用的代码被基础设施的代码污染后，也会影响可移植性。

在调用远程组件上的方法时，客户端需要封送（marshal）调用信息，管理网络链接，并通过网络传递调用请求、处理请求结果和错误。而本地方法调用并不需要这些活动。如果客户端应用直接处理这些问题，它们将依赖于特定的网络协议和IPC机制，那将降低它们在其他部署场景和应用中的可移植性和可重用性。而且，也会使得客户端开发者无法关注于他们的主要任务——正确而高效地实现应用的功能。

因此，创建一个Requestor，它负责创建和处理请求消息，并将其发送给远程组件（见图10-11）。

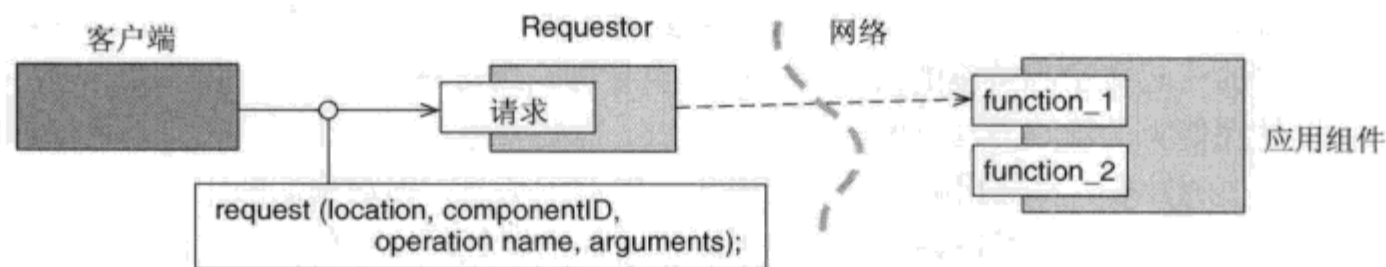


图 10-11

希望访问远程组件的客户端需要给Requestor提供远程组件的位置和引用，以及被调用的操作和它的参数等信息。Requestor利用这些信息构造一个请求消息，然后通过网络将它发送到远程组件。



在分布式系统中，Requestor为应用逻辑屏蔽了客户端网络，以及IPC活动与任务的诸多细节。

Requestor可以将它的一些子任务委托给其他的组件。Marshaler [VKZ04] 可以将一个具体的服务请求序列化为请求Message (245)，以及将包含了结果的消息反序列化为一个具体的响应。Client Request Handler (143) 可以管理连接并封装特定的IPC机制，因此能够简化跨网络的请求消息的发送与结果消息的接收。

一些应用需要Requestor完成额外的工作，比如添加安全令牌，Interceptor (260) 可以通过提供统一的接口来封装这项任务。通常，Marshaler、Client Request Handler和Interceptor在操作客户端请求处理的各个方面时，不会影响到Requestor关于创建、处理和发送请求的核心算法。Absolute Object Reference [VKZ04]可以封装远程目标组件的位置和标识信息。与前面类似，当过程中出现了失败，无法由Requestor透明地处理时，将会返回给客户端一个Remoting Error [VKZ04]。

通常，Requestor有3种部署方式[SMFG00]。最简单的是，为同一个节点上的所有客户端线程或者进程分配一个Requestor。然而，当有更多的客户端访问Requestor时，它就可能会成为系统吞吐量和可伸缩性的瓶颈。为了克服这个缺点，可以为每个客户分配独立的Requestor，或者由若干个客户端共享一个Requestor。由多个并发客户端共享的Requestor必须是同步的。为Requestor提供一个Thread-Safe Interface(224)，这种方式实现简单，但粒度较粗，因为它强制在Requestor的接口级别上进行同步，而方法中可能只有很少的片段属于临界区。在这种情况下，我们可以选择通过Strategized Locking (226)实现同步。将Requestor实现为Monitor Object (214)，可以支持并发控制多个同时访问Requestor的用户的协作。如果每个客户端应用都拥有多个Requestor，它们在内部使

用共享资源时必须做到同步，这些资源包括连接和请求对象的缓存等。

10.10 Invoker**

在构建服务器端Broker (137) 基础设施的时候……我们必须提供一种能够从网络中接收方法调用，并将其分派给远程组件的方法。



为了将从客户端接收到的数据转换为对远程组件上的特定方法的调用，服务器端需要执行大量管理和基础设施的任务。在每一个服务组件中重复地实现这些任务是非常繁琐的，而且容易出错，同时，应用的代码被基础设施的代码污染后，也会影响到可移植性。

要调用组件实现上的特定方法来响应客户端请求，需要服务器完成下面这些任务：管理网络连接，接收连接中的数据，将数据解封送以获取相关的调用信息，确定预期的组件实现，调用该组件上相应的方法，将结果或者错误返回。如果远程组件直接处理这些问题，它们将与特定的网络协议和IPC机制紧密地绑定到一起，这将降低它们在其他部署场景和应用中的可移植性和可重用性。而且，服务开发者也无法关注于他们的主要任务——正确而高效地实现应用的功能。

因此，创建一个Invoker，它可以将来自远程客户端的请求消息的接收和分派封装在组件实现的某个特定方法中（见图10-12）。

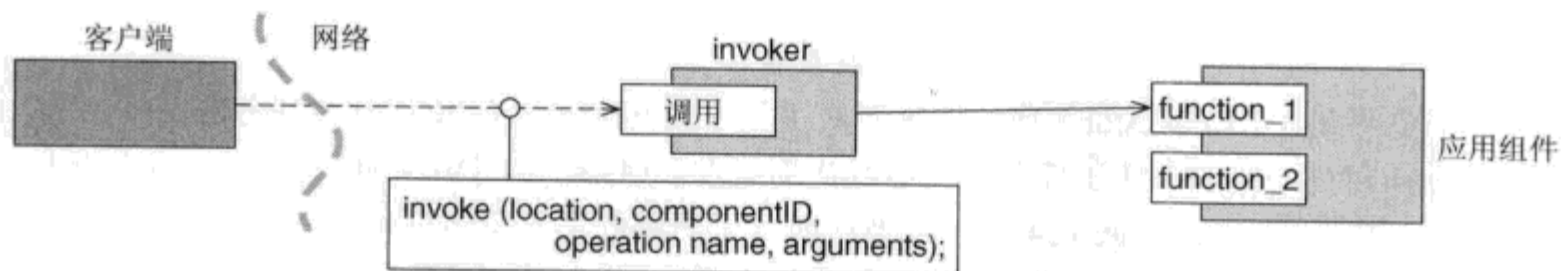


图 10-12

Invoker会监听网络连接，等待来自远程客户端的请求消息，当请求消息到达时将其接收下来，对消息进行解封送，判断应该调用哪个组件实现的哪个方法以及方法的参数，然后把这个方法分派到指定的组件上。



Invoker可以为分布式系统中的应用逻辑屏蔽掉服务器端网络以及IPC任务和活动的诸多细节。

Invoker可以将它多个子活动的职责委托给其他组件。Marshaler [VKZ04]可以将消息Message (245) 反序列化为一个具体的服务，再将相应的服务结果序列化到结果Messages中。Server Request Handler (144) 能够管理连接，并封装特定的IPC机制，从而简化了跨越网络接收请求和发送结果的过程。

有些应用还需要额外的调用活动，比如，解释嵌入的安全令牌——Interceptor (260) 可以通过统一的接口封装这个操作。通常，Marshaler、Server Request Handler和Interceptor在操作服务器端请求处理的各个方面时，不会影响到调用者关于接收和分派客户端请求的核心算法。

Absolute Object Reference [VKZ04]封装了特定组件实现的标识信息。当过程中出现了失败,无法由Invoker透明地处理时,将会返回给发送请求的客户端一个Remoting Error [VKZ04]。如果Invoker无法找到组件的实现(比如,组件已被重新部署或者组件在机器高负载的情况下暂时不可用),它可以把调用委托给Location Forwarder [VKZ04]。

Invoker有多种部署方式,最简单的是为服务应用中的全部组件只部署一个Invoker。但是,随着Invoker能够访问的组件越来越多,它就可能会成为吞吐量和可伸缩性的瓶颈。为了克服这个缺点,应该让几个远程组件共享一个Invoker,或者每个组件都有它自己的Invoker。

10.11 Client Request Handler**

在开发Requestor (140)的时候……我们必须将请求发送到网络,并且从网络获取回复。



发送客户端请求和从网络接收回复涉及到各种底层的IPC任务,比如连接管理、超时处理和错误侦测等。如果为每一个客户端都独立地编写和执行这些任务,这样使用网络和终端系统资源的方式是十分低效的。

对于分布式应用来说,访问网络的客户端越多,就有越多的请求和响应需要同时处理,也就越需要高效地管理网络资源,以达到预期的服务质量。例如,网络连接和带宽都是有穷资源,所有用户必须正确地共享和使用它们,这样才能确保延迟和抖动在一个可接受的范围内。另外,为每一个组件都单独编写错误检测和超时处理任务,将会导致代码的重复,而且应用会被不可移植的网络代码所污染。

因此,提供一个特化(specialized)的Client Request Handler,它可以代表客户端组件封装和执行所有的IPC任务。客户组件会从网络接收请求,并通过网络发送响应(见图10-13)。

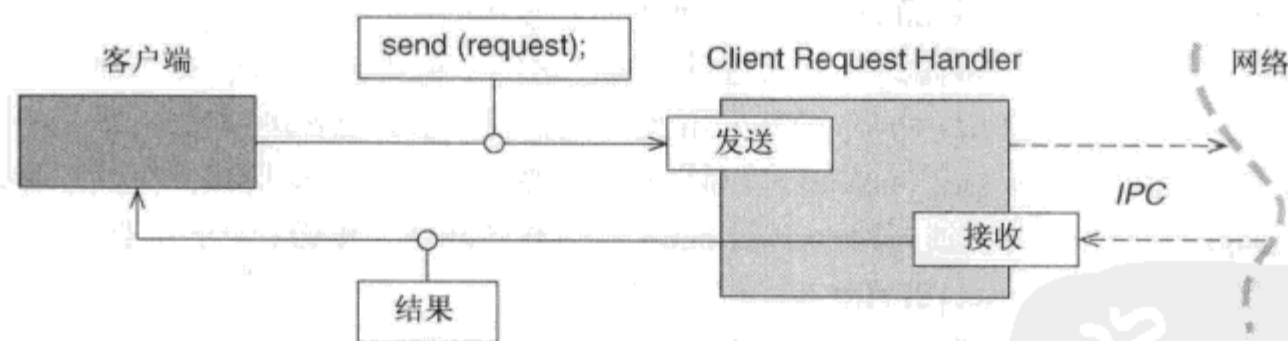


图 10-13

Client Request Handler的职责包括创建连接、发送请求和分派结果、处理超时和错误,这些都需要特定IPC机制的协助。另外,Client Request Handler还要负责网络和计算资源的高效管理和利用,比如网络连接、内存和线程。



在Client Request Handler内部对所有的客户端的网络资源实施集中化的执行和管理,可以提高分布式应用的服务质量,尤其是延迟时间、吞吐量、可伸缩性和资源的利用。客户端在向远程组件发送请求时,由于对特定的IPC机制进行了封装,因此通信过程对客户端是透明的。

为了创建一个到远程组件的实际连接, Client Request Handler实现了Acceptor-Connector (154) 中的Connector角色, 它可以使网络连接创建策略的变化独立于Client Request Handler的其他职责。如果Client Request Handler被多个并发的客户所共享, 那么Connector必须同步。为Connector提供一个Thread-Safe Interface (224), 这种方式实现简单, 但粒度较粗, 因为它强制在Connector的接口层面上进行同步, 而方法中可能只有很小的一部分属于临界区。在这种情况下, 可以考虑使用Strategized Locking (226) 来对同步机制进行参数化。将Connector实现为Monitor Object (214), 则可以在多个客户端同时访问Connector的时候, 支持协作并发控制。

由Connector创建的连接可以封装在连接处理程序中, 后者扮演了Acceptor-Connector中Service Handler的角色。这种设计将连接视为一阶实体 (First-class Entity), 这样就可以支持对特定于连接的状态进行高效维护, 同时可以处理产生于连接过程中的Remoting Errors [VKZ04]。同时也可以支持可伸缩性, 因为每一个连接处理程序都要运行在它自己的线程中, 所以能同时处理来自多个客户端的请求, 并同时回复多个客户端。

连接处理程序可以实现同步的或者异步的通信策略。同步通信能够简化客户端的编程模型, 但是也会降低性能和吞吐量, 而异步通信则恰恰相反。有4种模式有助于实现异步通信模型——Fire and Forget、Sync with Server、Poll Object和Result Callback [VKZ04]。这4种模式为应对下面3个问题给出了4种不同的解决方案, 这3个问题是: 结果是否需要发送给客户端; 客户端是否需要接收确认信息; 向客户端发出结果之后, 客户端是自主获取这个结果, 还是需要回调函数的通知。

如果客户端希望得到结果或者确认, 那么连接处理程序可以利用超时机制来检测异步通信中潜在的失败。它可以注册一个Reactor (150) 或者Proactor (152), 以便在结果到达的时候收到通知, 具体的选择取决于连接处理程序处理数据的机制是连续串行的, 还是由中断驱动的。

Connector和Client Request Handler的连接处理程序所使用的IPC机制, 可以用Protocol Plug-in [VKZ04] 或者一系列Wrapper Facade (269) 来进行封装。这两个模式都可以将IPC机制的细节隐藏到统一的、平台无关的接口之后。Protocol Plug-in允许运行时 (重) 配置IPC机制, 但是会导致一些运行时的开销。相反, Wrapper Facade可以避免运行时的开销, 却只支持编译时的配置。请求被封装在Message (245) 的内部, 通过Message Channel (130) 在网络上传递。如果需要考虑安全性, IPC机制应该使用Secure Channel [SFHBS06]来传送请求。

Object Manager (291) 可以通过缓存机制提高Client Request Handler的性能。例如, 对于不再需要的连接, 我们不必立刻将其销毁, 而是可以让它在一段预定的时间内保持“Alive”, 在它所连接的客户端和服务之间再次需要交互的时候重用。

调用的结果, 以及无法由Client Request Handler处理的Remoting Errors [VKZ04], 都会返回给发送相应请求的客户端。

10.12 Server Request Handler**

在开发Invoker (142) 的时候……我们必须跨越网络接收请求和发送响应。



从网络接收客户端请求和发送响应，这些任务涉及各种底层的IPC任务，包括连接管理、超时处理，以及错误侦测。如果为每一个客户端都独立地编写和执行这些任务，这样使用网络和终端系统资源的方式是十分低效的。

分布式应用的服务部分必须要同时处理的远程请求和回复越多，也就越需要高效地管理和使用网络资源，以达到预期的服务质量。例如，网络连接和带宽都是受限资源，因此所有远程组件必须高效地共享和使用它们，以提供可以接受的性能、吞吐量和分布式系统服务器端的可伸缩性。另外，为每一个组件单独编写错误侦测和超时处理任务，将会导致代码的重复，而且应用会被不可移植的网络代码所污染。

因此，提供一个特化的Server Request Handler，它可以代表远程组件封装和执行所有的IPC任务，比如从网络接收请求，并通过网络发送响应（见图10-14）。

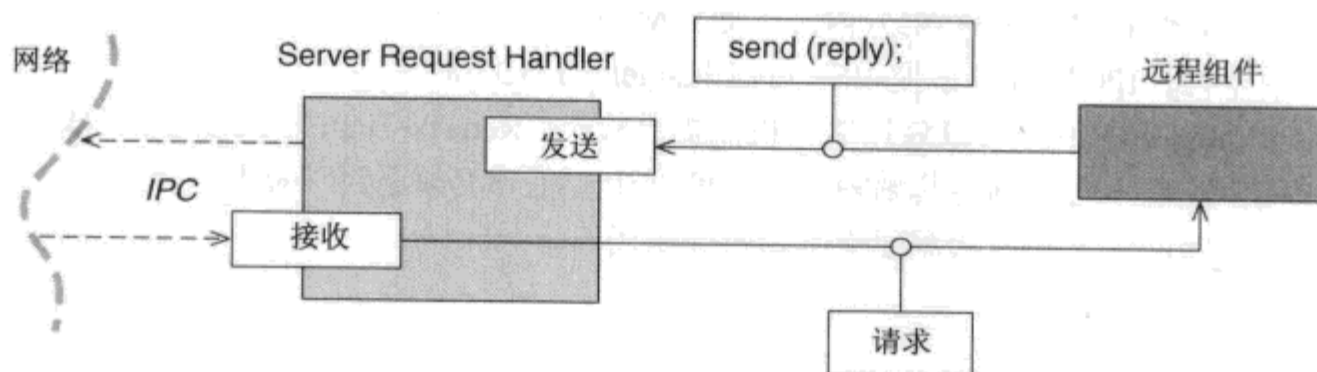


图 10-14

Server Request Handler的职责包括创建连接、接收和分派请求、发送执行结果和处理错误，这些都需要特定IPC机制的协助。另外，Server Request Handler还要负责网络和计算资源的高效管理和利用，比如网络连接、内存和线程。



在Server Request Handler内部对所有的服务器端的网络资源实施集中化的管理，可以提高分布式应用的服务质量，尤其是延迟时间、吞吐量、可伸缩性和资源的利用。应用组件在接受请求和将响应发回给远程客户端时，由于对特定的IPC机制进行了封装，因此通信过程对应用组件是透明的。

Server Request Handler需要一个事件处理基础设施，它会监听网络中到来的连接请求，建立请求的连接，将服务请求分派到合适的应用组件的方法上。这个基础设施必须可以同时接受和处理多个连接和服务请求，这样才能得到适当的延迟时间、吞吐量和可伸缩性。它的核心可以实现为Reactor (150)或者Proactor (152)，这取决于处理接收事件的机制是串行的，还是由中断驱动的。

Reactor或者Proactor的事件处理程序可以实现为Acceptor-Connector (154)，以此将连接的创建同请求和回复的处理机制分离开。使用一个专用的Acceptor来监听网络，等待连接请求的发生，接受请求，然后创建一个连接处理程序（用于封装新建立的连接）。该连接处理程序代表（通过它所封装的连接进行访问的）应用组件执行IPC操作。这种设计使得连接的创建方式和数据的传递策略可以彼此独立地变化。连接被视为一阶实体（First-class Entity），这确保我们可以有效地

维护与连接相关的状态，并处理连接过程中发生的任何Remoting Errors [VKZ04]。另外，可伸缩性也得到了支持——每个连接处理程序可以运行在它自己的线程中，这允许Server Request Handler可以并发地处理来自多个客户端的请求和响应。

连接处理程序可以实现同步的或者异步的通信策略。同步通信可以简化客户端的编程模型，但是会降低性能，异步通信则恰恰相反。Sync With Server模式[VKZ04]可以为匿名的通信模型返回确认信息。

Acceptor和Server Request Handler的连接处理程序所使用的特定的IPC机制，可以通过Protocol Plug-in [VKZ04] 或者一系列的Wrapper Facade (269) 进行封装。这两个模式都可以将IPC机制的细节隐藏到统一的、平台无关的接口背后。Protocol Plug-in允许在运行时对IPC机制进行（重）配置，但是会带来一些运行时的开销。相反，Wrapper Facades可以提高性能，但只能支持编译时的配置。请求被封装在Message (245) 内部，通过Message Channel (130) 在网络上进行传递。如果需要考虑安全性，IPC机制应该使用Secure Channel [SFHBS06]来传送请求。

Object Manager (291) 可以通过缓存机制提高Server Request Handler的性能。例如，对于不再需要的连接，我们不必立刻将其销毁，而是可以让它在一段预定的时间内保持“Alive”，在它所连接的客户端和服务之间再次需要交互的时候重用。





慕尼黑的出租车站
© Frank Buschmann

分布式计算的核心都是关于处理和响应网络中收到的事件的。因此本章给出了4个模式，来描述分布式网络系统中发起、接收、分离、分发和处理事件的不同方法。

也许中间件平台为应用提供了更复杂的通信模型，如请求/响应操作或异步消息，但分布式计算从根本上来说仍然是事件驱动的。事件驱动的软件所要面对挑战与一般的“自主”（self-directed）控制流软件有所不同[PLoPD1]。

- 事件的异步到达。事件驱动软件的行为在很大程度上都是由那些异步到达的外部或内部事件触发的。绝大部分事件必须被及时处理，即使是在系统负荷很大或在执行长时间服务的时候。如果没做到，响应时间就会受到影响，具有实时性限制的硬件设备会出现故障或者损坏数据。
- 多个事件的并发到达。通常事件驱动软件从多个独立的事件源接收事件，如I/O端口、传感器、键盘或鼠标、信号、定时器或异步软件组件等。因此，多个事件可能会同时到达应用。为了及时响应从任何事件源发来的任何事件，事件驱动软件必须能监听所有事件

源的事件。

- 事件到达的不确定性。尽管事件驱动软件通常很少能控制事件到来的顺序，但是它必须能及时处理事件，不管它们以什么样的顺序到达。需要按照特定顺序处理事件的软件必须能检查出错误的事件顺序，从而避免错误的状态转换。这种需求促使我们为事件驱动软件提供灵活有效的事件分离和分发基础设施。
- 多种事件类型。大多数事件驱动软件处理多种事件，每种事件类型都需要特定的行为。例如，Connect事件表示在两者之间建立连接的请求，而Data事件表示操作请求和请求的参数。将事件分发给正确的处理程序是事件处理框架的责任，这需要一个高效的机制来将事件分离到期望的处理程序上，并分发给正确的服务或操作来处理事件。
- 隐藏事件分离和分发的复杂性。用来检测、接收和分离事件的底层操作系统机制通常既繁琐又容易出错[SH03]。为了简化事件驱动软件的开发，我们需要较高层次的抽象，使得应用服务不必关心事件分离和分发的复杂性。

为了优雅而有效地应对上述挑战，事件驱动软件通常采用控制流倒置（inverted flow of control）的Layers (108) 架构[John97]。该架构中的每一层负责处理事件驱动运算中的某一特定方面，并向高层隐藏与该方面相关的复杂性。典型的事件驱动软件有以下3层。

- 事件源，如套接字（socket）[Ste98]在最底层，负责从各种硬件设备或操作系统的底层服务检测和接收事件。
- 它上面的一层是事件分离器（event demultiplexer），使用像WaitForMultipleObjects、GetQueuedCompletionStatus[Sol98]、select[Ste98]或者poll[Rago93]这样的函数来等待各种事件源发送的事件，然后再将事件分发给相应的事件处理程序回调。
- 事件处理程序和应用代码一起构成了第3层，作为对回调的响应在不同的应用中做出相应的处理。

尽管Layers方案通过单独处理每个关注点的方式将事件驱动软件的不同关注点分离开。但它并没有解释怎样在一系列驱动因素下来优化解决某一特定关注点的问题。例如，事件分离层本身并不能确保将事件高效简单地分离并分发到事件处理程序中。

分布式计算模式语言的4个事件处理模式有助于解决这一问题。它们为事件驱动的软件（event-driven software）中关键的事件分离和分发问题提供了高效、可扩展、可重用的解决方案。

- Reactor（反应器）模式 (150) [POSA2] 允许事件驱动软件分离和分发从一个或多个客户端传递到应用的服务请求。
- Proactor（前摄器）模式 (152) [POSA2] 允许事件驱动软件有效分离和分发由异步操作完成所触发的服务请求，从而获得性能和并发性的提升，避免它的某些不足。
- Acceptor-Connector（接受器-连接器）模式 (154) [POSA2] 将网络系统中互操作的对等服务之间的连接和初始化与连接和初始化之后的对等服务处理相分离。
- Asynchronous Completion Token（异步完成令牌）模式 (155) [POSA2] 允许事件驱动软件能高效地分离和处理异步服务调用的响应。

本章只关注与分布式计算相关的事件分离和分发的模式。其他领域事件驱动软件的模式，如

处理用户界面事件等，不包含在本章之内。

Reactor和Proactor模式定义了事件分离和分发的框架，可以用来为事件驱动应用检测、分离、分发和处理它们从网络上接收到的事件。尽管两个模式在相似的环境中解决本质上相同的问题，并使用相似的模式来实现其解决方案，但由于每个模式展现出的不同驱动因素，其所建议的具体事件处理框架也是不同的。

Reactor关注简化事件驱动软件的编写。它实现了一个被动的事件分离和分发模型，服务等待请求事件的到来，再通过不受间断的同步处理事件，从而作出反应。虽然这种模型在请求响应事件很短的情况下能很好的扩展，但对于长期服务则会引入性能损伤，因为同步执行这些服务会导致对其他请求服务的过度延时。相反，Proactor的设计是为了最大程度地提高事件驱动软件的性能。它实现了一个更主动的事件分离和分发模型，服务将其过程划分成多个自包含的部分并预先触发这些部分的异步执行。这种设计允许服务并发地执行，从而提高服务质量和吞吐量。

因此，Reactor和Proactor实际上并不是可替换的同等设计，而是互相补充的模式，根据编程简单性和性能作出折中。相对简单的事件驱动软件可以从基于Reactor的设计中受益，而Proactor提供了一个效率更高并更容易扩展的事件分离和分发模型。

图11-1描绘了Reactor和Proactor是怎样集成到我们模式语言中的。

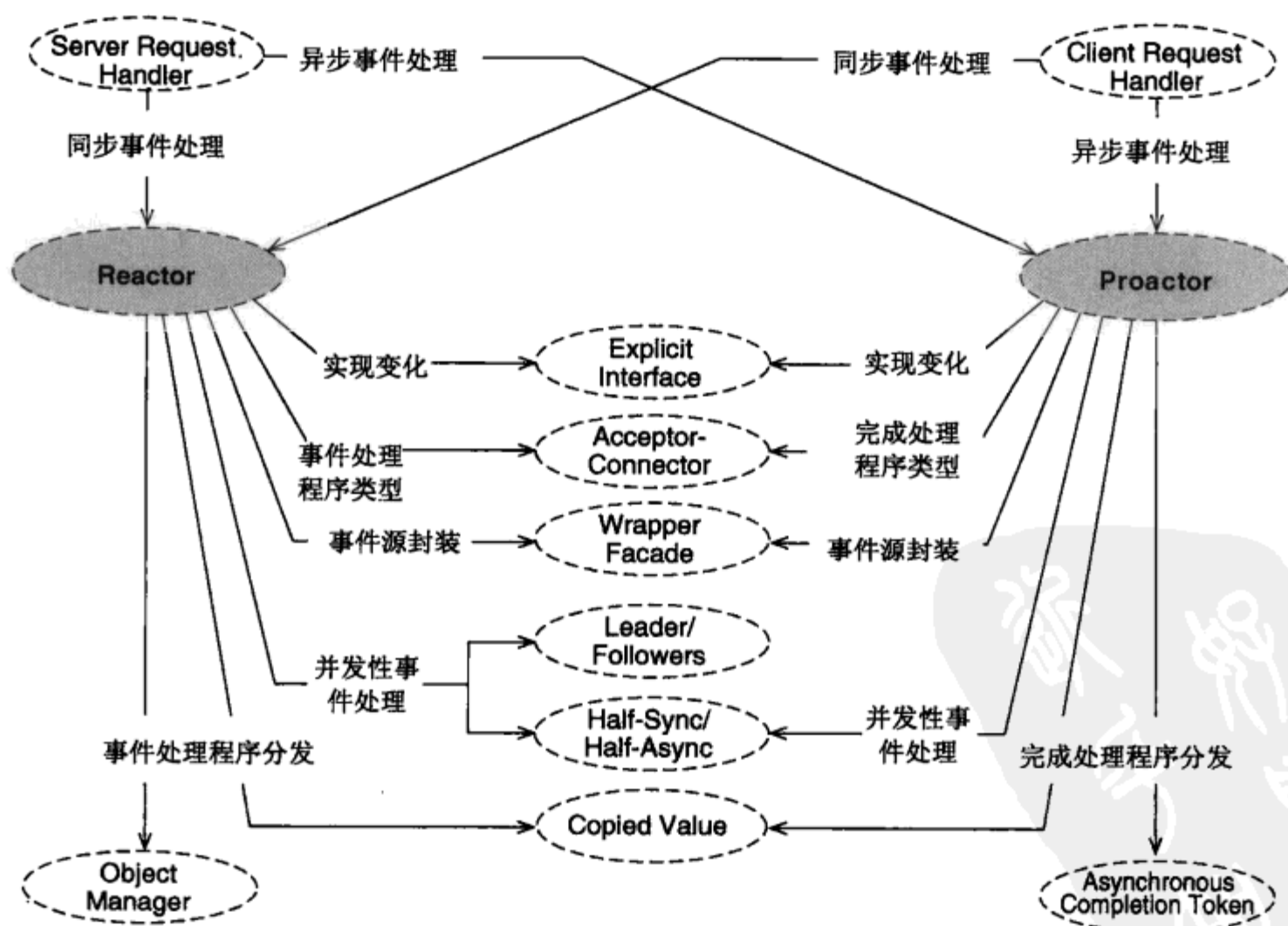


图 11-1

Acceptor-Connector和Asynchronous Completion Token帮助我们细化Reactor和Proactor引入的事件处理框架。本质上来说, Acceptor-Connector根据特定责任划分事件处理程序: 发起到远程对等处理程序的连接, 接受远端连接请求, 以及事件处理。这种分离支持连接建立及初始化行为和服务处理程序功能之间的独立性。此外, 它使得应用开发人员从处理底层连接管理的任务中解脱出来。Asynchronous Completion Token支持异步服务调用响应和相应请求之间的关联, 从而请求的发送者能在固定的时间内根据响应决定相应的行为。

图11-2描绘了Acceptor-Connector和Asynchronous Completion Token是怎样与我们语言中的其他模式联系起来的。

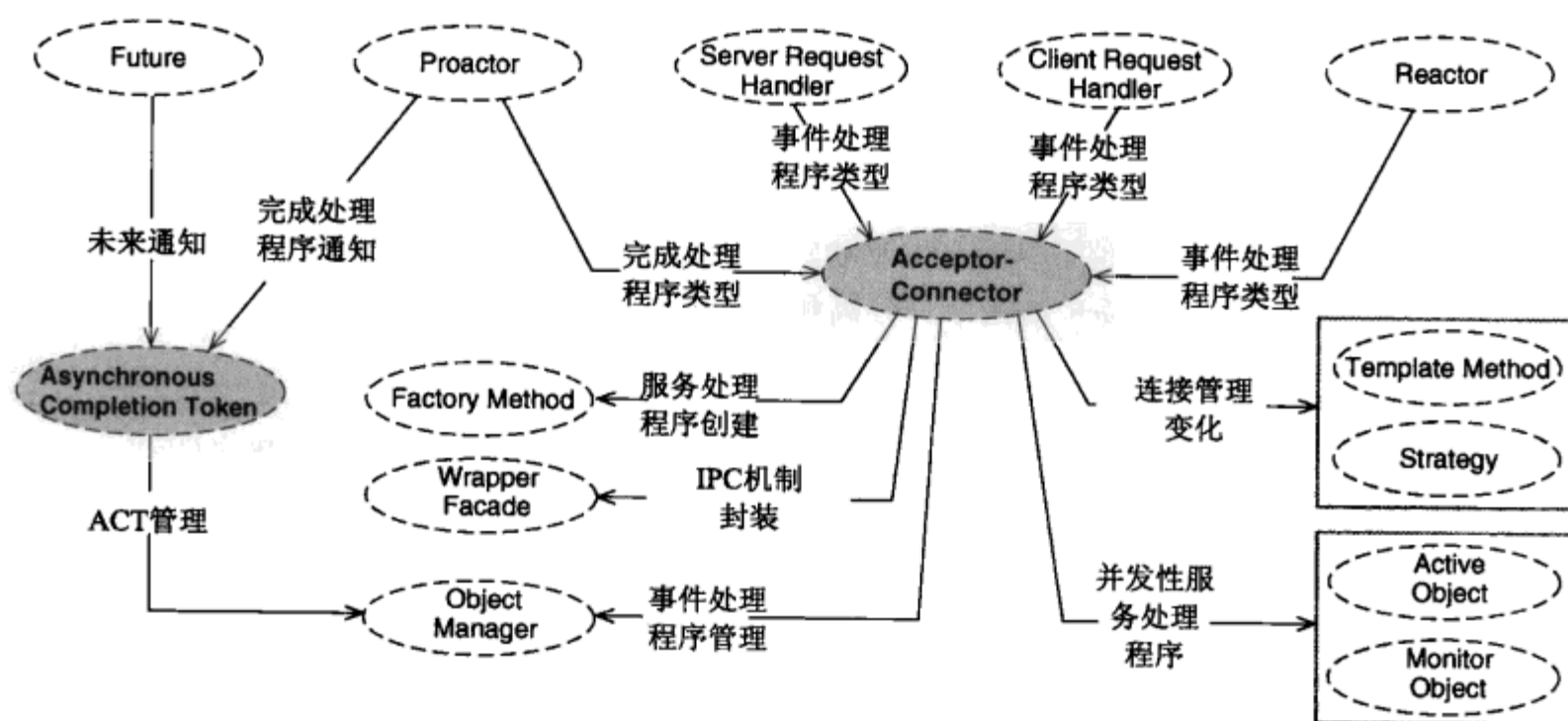


图 11-2

然而, 所有4种模式都能应用于更广泛的领域, 而不仅仅是处理网络事件。例如, Reactor和Proactor还能应用于分离和分发用户输入事件给用户界面元素。

11.1 Reactor**

在开发事件驱动软件, 或者Client Request Handler (143) 或Server Request Handler (144) 的时候……我们必须将与事件检测、分离和分发相关的框架行为与为事件提供服务的短时间组件分离开。



事件驱动软件经常从多个事件源接收服务请求, 并将事件分离并分发给事件处理程序以完成进一步的服务处理。另外, 多个事件还可能同时到达事件驱动应用。然而, 为了简化软件开发, 事件应当被串行同步处理。

灵活高效地处理同时从多个事件源到达的事件是很困难的。例如, 使用多线程等待事件发生会引入同步、上下文切换以及数据移动方面的额外开销。相反, 无限制地阻塞在单一的事件源上

则会妨碍对其他事件源的服务，降低对客户端的服务质量。此外，应当很容易将新的或改进的事件处理程序集成到事件处理框架中。

因此，提供一个事件处理框架，使其可以同时等待多个事件源上所出现的服务请求事件，但是每次只分离和分发一个事件给相应的事件处理程序以完成其服务（见图11-3）。

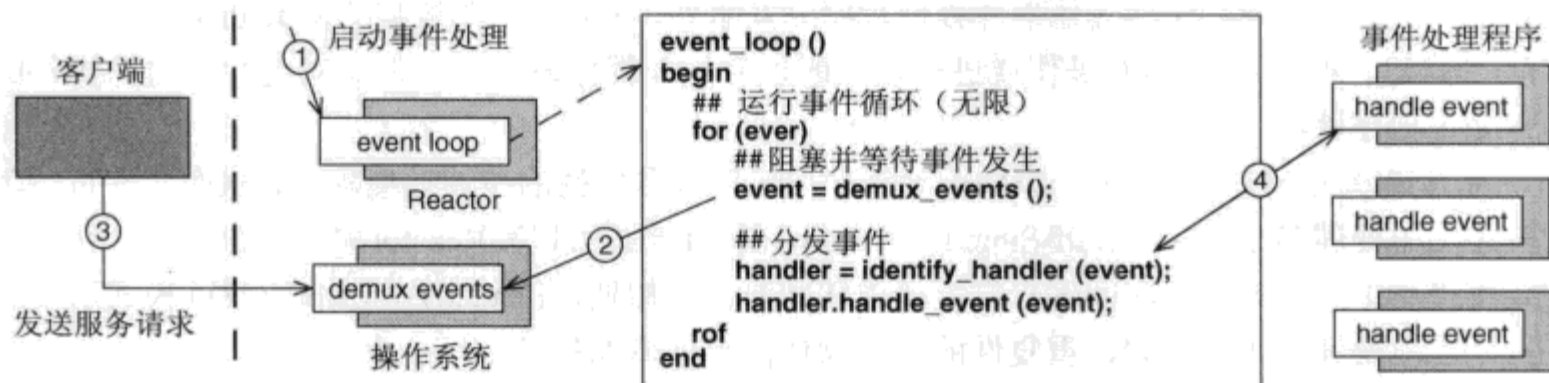


图 11-3

Reactor组件协调事件驱动应用的事件处理。它定义了一个事件循环，利用操作系统事件分离器在一系列事件源上同步等待服务请求事件的发生。通过将事件分离交给操作系统，Reactor能同时等待多个事件源而不需要在应用中采用多线程。当事件到达后，事件循环将事件逐个分发给相应的事件处理程序，后者实现了所要求应用功能。然后，每个事件处理程序再对“它的”事件作出同步反应，并执行相应的服务。



Reactor设计有几个好处。首先，操作系统事件分离机制可以在多个事件源上等待，并避免了多线程相关的性能开销和编程复杂性。其次，将软件事件循环封装在Reactor中使得服务事件处理程序不需要关心同步事件分离和分发框架的复杂性。第三，事件的串行化对应用组件是透明的，可以顺序地同步执行而不需要明确的加锁机制。

Reactor组件构成了事件驱动软件的核心，它封装了一个可重用的事件分离和分发框架。特别是Reactor定义了一个事件循环，利用底层操作系统提供的事件分离器，如select或WaitForMultipleObjects，在指定的多个事件源上等待服务请求事件的发生。调用事件处理程序阻塞处理程序的执行，直到从事件源接收到一个或多个事件，并且可以以非阻塞的方式处理这些事件。事件分离器将所有产生事件的事件源句柄返回给Reactor，再由Reactor将其以Copied Value (230) 方式逐一分发给处理程序，以同步响应和处理这些事件。

不同的Reactor实现经常需要平台提供不同的事件分离器。在这种情况下，可能需要Explicit Interface (163) 来分开Reactor的接口和实现。事件处理程序通常采用Acceptor-Connector (154) 方式组织，其中服务处理程序提供领域相关的功能，而Acceptor和Connector为服务处理程序建立连接。可以通过为所有事件处理程序定义公共的Explicit Interface，规定处理服务请求事件可用的一套操作，从而支持可扩展的事件处理框架。这种设计使得Reactor和特定服务的规范和逻辑之间的耦合最小，所有服务都使用一个公共的事件处理程序接口。

处理服务请求事件通常需要事件处理程序在事件到达时对事件源进行额外的I/O操作，例如

读取客户端请求相关的参数值，或向发出事件的客户端返回结果。可以使用 Wrapper Facade (269) 简化事件处理程序和事件源之间的通信，并避免对平台相关的 I/O 函数的依赖。Reactor 内的 I/O 处理和事件处理程序通常通过 Object Manager (291) 存取。

事件处理程序直到处理完服务请求事件才将控制权返回给 Reactor。因此，如果事件处理程序阻塞了一段时间，则无法给其他事件处理程序分发事件。所以说，单线程的 Reactor 配置对进行短时间的服务、不会阻塞在 I/O 操作或进程锁上的事件处理程序最适合，但是对进行长时间操作的事件处理程序就不可行了。

为了减轻这一缺陷的影响，事件驱动软件可以实现并发性的事件处理程序，从而允许事件驱动应用同时处理多个事件。Half-Sync/Half-Async (209) 模式可以和 Reactor 模式联合工作，在单独的线程控制中并发处理长时间客户端请求和回复。类似地，Leader/Followers (211) 模式适用于使用线程池处理大量短时间、重复性的原子操作的事件驱动软件。

11.2 Proactor*

在开发事件驱动软件，或者 Client Request Handler (143) 或 Server Request Handler (144) 的时候……我们必须将与事件检测、分离和分发相关的框架行为与为事件提供服务的长时间组件分离开。



为了得到所需要的性能和吞吐量，事件驱动应用往往需要能同时处理多个事件。然而为了避免同步、上下文切换以及数据移动方面的开销，我们可能并不希望采用多线程。

然而，服务处理不应当因为事件源的长周期活动而过度延迟，比如等待远程客户端的服务请求事件，或者与客户端或其他组件（如数据库）进行 I/O 操作等。性能和吞吐量应当尽可能大。此外，应当易于将新的或改进的组件集成到现有的事件处理框架中。

因此，将应用功能分解为在事件源上进行活动的异步操作和使用异步操作结果实现应用服务逻辑的完成处理程序（Completion Handler）。由操作系统执行异步操作，但在应用的控制线程中执行完成处理程序（见图 11-4）。

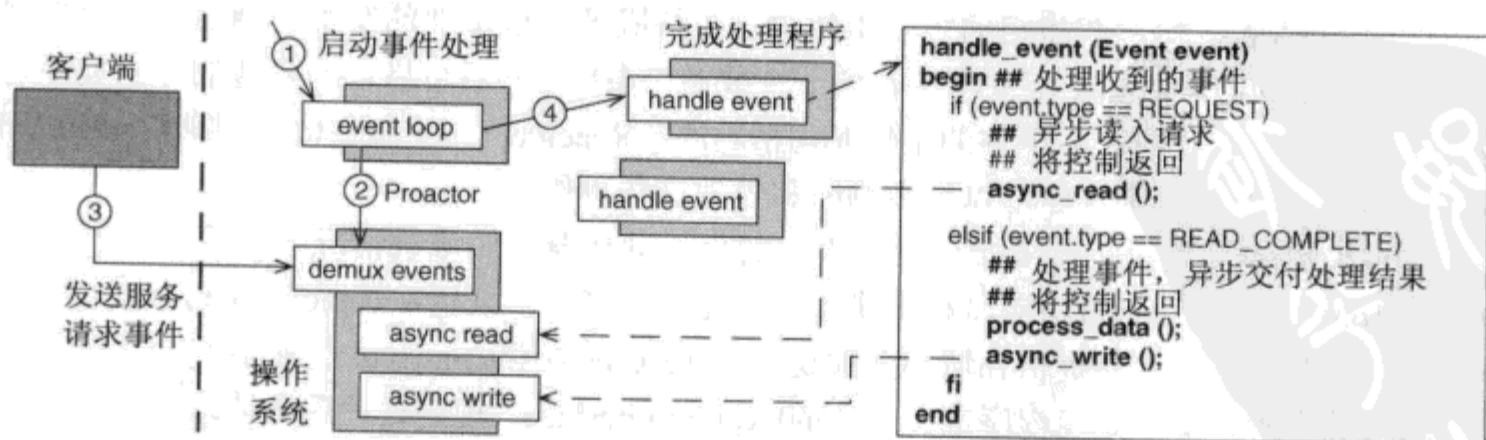


图 11-4

Proactor 组件协调完成处理程序和操作系统之间的配合。它定义一个事件循环，以使用操作

系统的事件分离器同步等待表明异步操作完成的事件的发生。开始所有完成处理程序“前摄性地”(proactively)调用异步操作以等待服务请求事件到达,然后在Proactor中运行事件循环。当这样的事件到达后,Proactor把完成的异步操作的结果分发给相应的完成处理程序。该处理程序再继续执行,比如调用另一个异步操作等。



Proactor事件处理基础设施允许在单一应用线程中同时执行多个长时间服务——只要完成处理程序不在同一时刻操作同一资源,从而避免多线程开销,如同步、上下文切换以及数据移动等,增强了事件驱动软件的性能和吞吐量。将软件事件循环封装在Proactor中还使得完成处理程序不需要关心异步事件分离和分发基础设施的复杂性。

要实现Proactor方式,大多数现代操作系统提供了异步操作,如Real-time POSIX中的aio_* API[POSIX95],以及Windows中的Overlapped I/O[Sol98]。操作系统在句柄指定的事件源上执行这些异步操作而不阻塞调用者。完成处理程序可以使用这些操作将长时间的I/O操作交由操作系统完成,从而使得处理程序能处理其他请求,直到操作完成。Proactor的不足之处在于它依赖于操作系统对异步I/O的支持和高效运行。

考虑到效率,完成处理程序和异步操作之间的协作通常是基于Asynchronous Completion Token (ACT) (155)的。当一个完成处理程序激发了一个异步操作,它传入一个ACT以包含调用处理程序的唯一标识,以及所执行操作对应事件源的句柄。当异步操作完成时,它将结果填入ACT中,并且由操作系统在相应的事件源上产生一个完成事件,以表明操作结束。该完成事件同样包含相应的ACT。

完成事件通过Proactor返回给完成处理程序,Proactor的事件循环使用操作系统提供的事件分离器,如Windows API[Sol98]中的GetQueuedCompletionStatus。该事件分离器等待句柄所指定的事件源上产生的完成事件。当完成事件产生时,事件分离器将其返回给Proactor,再由Proactor利用其ACT将每个事件以Copied Value (230)分发给相应的完成处理程序,以处理异步操作的结果。一旦完成处理程序调用另一个异步操作,控制返回给Proactor,从而等待并分发下一个完成事件。

不同的平台经常要求采用不同的Proactor实现,因此Proactor的接口应当通过Explicit Interface (163)和实现分离开。完成处理程序通常设计成Acceptor-Connector (154)方式,其中服务处理程序提供领域相关的功能,Acceptor和Connector负责为服务处理程序异步建立连接。可以为所有完成处理程序定义一个公共的Explicit Interface来指定可用于处理服务请求事件的操作集,从而支持可扩展的事件处理框架。这种设计使得Reactor和特定服务的规范和逻辑之间的耦合最小,所有服务都使用一个公共的事件处理程序接口。

处理服务请求事件通常需要完成处理程序在事件到达的事件源上进行额外的I/O操作,如异步读取客户端请求相关的参数值,或向发出事件的客户端返回结果。可以使用Wrapper Facade (269)简化事件处理程序和事件源之间的通信,并避免对平台相关的I/O函数的依赖。

Half-Sync/Half-Async (209)模式可以和Reactor模式联合工作,同步并顺序地处理长时间的客户端请求和回复,从而既可以简化应用编程,又不会降低Proactor事件处理框架的性能。

11.3 Acceptor-Connector**

在面向连接的网络系统中，要实现事件处理程序，例如Reactor (150) 架构中的事件处理程序或Proactor (152) 架构中的完成处理程序，或者在设计Client Request Handler (143) 或Server Request Handler (144) 的时候……我们希望将基础设施行为和应用相关的行为分离开，前者负责建立连接和初始化事件处理程序，后者负责处理程序的执行过程。



在网络系统中，对等事件处理程序（peer event handler）之间执行功能之前，必须先彼此建立连接并初始化。然而，对等事件处理程序的连接建立和初始化代码在很大程度上与其完成的功能无关。

更复杂的是，事件处理程序的应用功能通常比连接和初始化策略变化得更频繁。此外，事件处理程序可能动态改变其在连接中所扮演的角色：在某种情形下它主动发起到远端的连接，而另一种情形下它被动接受远端的连接请求。

因此，将网络中对等事件处理程序的连接和初始化同接下来进行的处理分离开（见图11-5）。

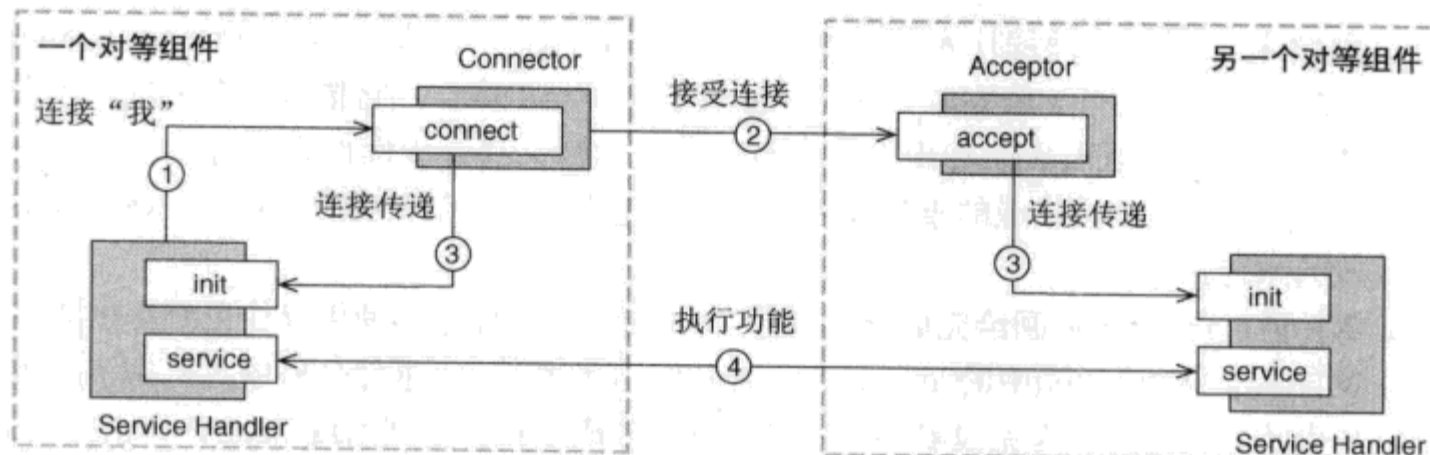


图 11-5

客户端服务处理程序可以通过调用本地Connector（可以主动发起到远程对等端新连接的工厂）发起到远程服务处理程序的连接。Connector发送请求到相应的Acceptor（位于服务器端的被动接受远端连接请求的工厂）。当这些工厂建立两端之间的连接后，它们初始化相关的服务处理程序并将该连接传递给处理程序。然后处理程序利用这个连接互相合作，执行其应用功能。



Acceptor-Connector方式将连接建立封装在单独的Acceptor和Connector组件中，从而使得服务处理程序不必关心底层网络编程基础设施的复杂性。此外，连接建立和初始化行为可以独立改变而不影响服务处理程序的功能。然而，需要注意的是，对于只连到一个服务器并且使用单一网络编程接口完成服务的简单客户端应用来说，Acceptor-Connector方式可能会增加不必要的复杂性。

在设计Acceptor的时候，我们可以利用事件分离器被动监听连接请求。如果Acceptor-Connector方式是建立在Reactor或Proactor的基础之上，Acceptor可以利用事件处理框架所提供的事件分离器。当连接请求到达时，事件分离器将请求分发给Acceptor，Acceptor执行三个步骤与

发出请求的远端建立连接。首先，由Factory Method (313) 创建要连接到远端的服务处理程序实例；其次，Connector和远端建立连接；第三，将连接传递给相关的服务处理程序并对其完成初始化工作。

总之，Connector工厂能为服务处理程序提供同步或异步的连接建立策略。如果连接建立的延迟较小，同时服务处理程序必须按固定顺序初始化，而且服务处理程序可以采用每个连接一个线程的模式连接远端，同步连接建立的方式是最有效的。反之则应该使用异步连接策略。可以通过将Connector划分为一个connect方法和一个complete方法来透明支持同步和异步两种连接建立方式，服务处理程序调用connect方法来主动建立连接，complete方法在连接建立之后将其传递给请求该连接的服务处理程序[POSA2]。

Connector和Acceptor工厂中的每个活动都可以借助于Template Method (265) 或Strategy (266) 来实现。这样设计使得连接建立和初始化策略的灵活改进和替换对于服务处理程序来说是透明的。如果这种灵活性是在编译时需要的，那么Template Methods最适合，而Strategies支持在运行时对Acceptor和Connector进行配置和重新配置。在C++中，Strategy也可以采用编译时策略（compile-time policy）来表现，这两种方式的区别在于是基于继承还是基于委托（delegation-based）的方式，与绑定时间无关。

Connector组件用来支持异步连接建立的I/O句柄和事件处理程序通常通过Object Manager (291) 存取。

Acceptor和Connector工厂完成连接建立功能，以及服务处理程序完成应用功能是通过采用IPC机制和对等端交换消息实现的。将IPC机制封装在一套Wrapper Facade (269) 中以确保在Acceptor和Connector工厂及服务处理程序中的正确使用和可移植性。

由于可以同时处理多个事件，并发性服务处理程序能改善事件驱动应用的吞吐量。如果并发性服务处理程序表示粗粒度的组件，则可以实现为Active Object (212)；如果是细粒度（分布式）对象，则可以实现为Monitor Object (214)。

11.4 Asynchronous Completion Token**

当开发Proactor (152) 框架时，应用中使用Future (223) 设计或者总体上使用异步通信……我们必须高效分离并处理服务器调用异步操作的响应事件。



对于一个双向操作来说，客户端以异步的方式发起调用，响应信息则通过“完成事件”（completion event）来返回。然而客户端并不会在调用操作后阻塞。因此，客户端的状态在完成事件到达时和发起调用时可能有所不同。

为了正确执行，客户端必须在合适的环境下处理操作结果。为了增强性能，客户端还应当花尽可能少的时间确定怎样处理异步操作的响应。此外，如果客户端调用多个异步操作，则响应到达的顺序可能和调用操作的顺序不完全相同。

因此，客户端在每个异步操作调用的同时传送一个异步完成令牌（Asynchronous Completion Token, ACT），以包含其所需要的用于确定怎样处理操作响应的最少信息（见图11-6）。

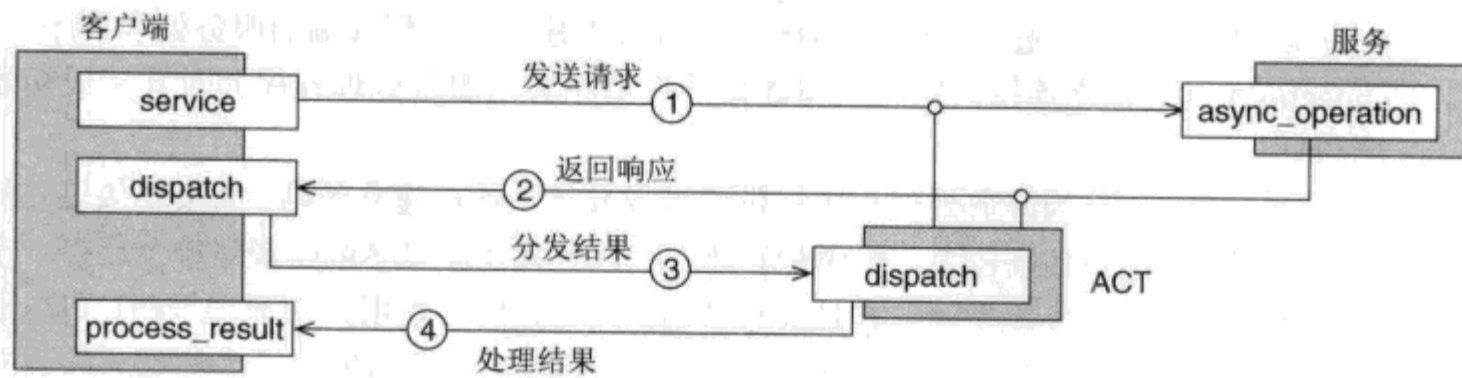


图 11-6

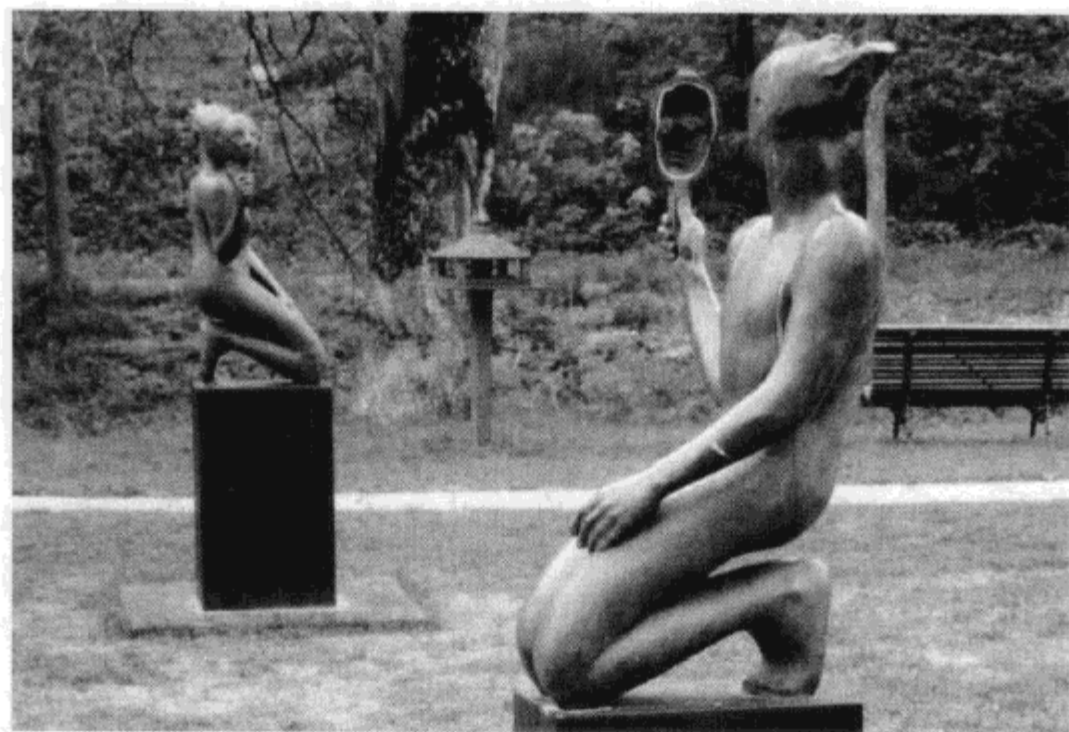
ACT将每个异步操作调用和完成时需要客户端执行的特定行为联系起来。当客户端异步触发服务操作时，它还将ACT传递给该服务。服务在执行操作时保留该ACT，但并不修改它。一旦操作完成，将ACT和结果一起返回给客户端。客户端使用ACT来高效而明确地确定要执行的行为，作为对操作完成的响应。特别地，客户端使用ACT将控制流分离并分发到负责处理操作结果的方法或处理程序上。



用Asynchronous Completion Tokens来分发和处理异步操作结果，使得客户端可以在常数时间内指定用以处理结果的方法和处理程序。此外，ACT只需要使用最小的存储空间将响应和对应的请求匹配起来。但另一方面，使用ACT会使得应用更容易受到意外错误或恶意攻击的侵害，因为它假定异步操作不关心且不修改其数据结构。

实现Asynchronous Completion Token的方式有很多。一种方法是系统级语言（如C++）所特有的，将负责处理相关异步操作结果的方法或处理程序的内存地址作为ACT的值。当ACT返回时，指针被转换为合适的调用方法或处理程序。然而，如果方法或者处理程序在虚拟内存中被重新映射了（如果对象是在“内存映射地址空间”（memory-mapped address space）分配的，就有可能出现这种情况），使用内存地址作为ACT可能引入一些诡异的错误。如果将内存地址作为ACT有风险，我们可以使用代理标识（proxy identifier），如对象引用或表中的索引。将“真正的”ACT保留在客户端，如Object Manager (291) 中，当服务将ACT的代理标识返回时，通过它得到“真正的”ACT并加以处理。

为了可靠地回收ACT——即使当异步操作失败时，可使用Object Manager控制客户端内所有ACT的生命周期，例如在调用返回错误时或超出一段设定的时间后将其删除。



Hilde Mahlum的作品Maskebærere（持面具者）在挪威奥斯陆的Bærum Verk^①展出
©Kevlin Henney

确定组件接口是软件项目中非常重要的活动。接口应当清楚地反映组件的职责和使用协议，为客户端提供有意义的服务，并且让客户端避免由于组件实现的变动和升级而受到影响。否则，组件将会变得难以使用，而且彼此之间的协作也会变得很复杂。本章给出了11个模式来具体说明怎样设计具有良好定义的接口，以满足上述要求。

接口是组件的“名片”，它应当让客户端了解组件的职责、服务和使用协议，并且应当便于客户端与组件间正确有效地协作。因此，设计和定义可用的、有意义的组件接口是成功软件开发的关键所在。此外，能否高效地使用基于组件开发（Component-Based Development）[Szy02]和面向服务架构（SOA）[Kaye03]等软件开发和部署方案，严重依赖于可用服务和组件接口的质量。最终，不恰当的接口不仅会降低组件的可用性，还会增加应用的结构复杂性，从而使得组件难于

① 位于奥斯陆附近阿克什胡斯（Akershus）小镇的一个村庄。——译者注

理解、维护和升级[Bus03]。

要为应用中的组件设计高质量的接口，开发人员需要解决以下具有挑战性的、有时甚至互相冲突的问题。

- **组件职责和约定规范。** 组件——特别是多用途组件——的开发人员通常无法控制哪些应用会使用他们的组件。对于这些应用“如何”使用其组件，他们能控制的也十分有限。这就要求组件开发人员必须详细地指明组件的具体职责、功能和使用协议。不同的组件职责必须清楚地划分开，以避免客户端对组件的目的和使用产生困惑。组件应当在所有适用的场合都易于使用（重用），还要防止恶意使用。
- **质量属性。** 如果一个组件在使用时能保证自己的行为正确，并在错误发生时保持健壮性，它就好比那些不具备这些质量属性的组件容易使用和重用。另外，在并发和分布式应用部署环境下的质量属性也必须考虑，包括远程调用的性能、共享并发组件的同步、组件的正确创建和回收，以及组件功能的安全访问。这些考虑不仅仅局限于实现细节，它们往往需要反映在设计和接口规范中。理想情况下，我们应该在接口中约定如何高效正确地使用组件。
- **可表达性和简洁性。** 组件接口越简洁、表达能力越强，就越容易使用。因此，具体方法应当在不破坏组件封装性的前提下，尽量反映客户端的使用意图。理想情况下，简单通用的功能应当易于使用，而复杂的使用场景应当也可以应付，尽管也许稍微麻烦一些。
- **松耦合和稳定性。** 客户端通常对所使用组件的内部设计和实现不感兴趣，它们也不应当依赖于它们所用不到的组件接口和使用协议。此外，接口版本和稳定性是成功使用组件的关键因素，客户端没有使用的方法签名发生变化时不应当影响到客户端，组件扩展了新的功能和角色时也不应当影响到不相关的客户端。总之，组件所公开的接口应当保持稳定，使得客户端尽可能地不受组件内部实现的影响。

这一特性对分布式系统中的组件非常重要，对于设计应用于面向服务架构的系统就更为重要了。在这样的系统中，我们很难预先确定组件的客户端，而且客户端的实现并不总是能在组件接口变化时跟着一起改变。此外，许多客户端可能在运行时访问组件，这使得在线升级更加困难。因此组件之间的松耦合，加上接口的稳定性，是构造可持续且支持可控改进的软件系统的关键所在。

解决这些挑战相当困难，而且需要深入的领域知识、设计能力和实现技巧。

将组件扩展到分布式系统则使得接口设计更具有挑战性，这是因为下列原因。

- **组件分布性。** 组件的实现部分可能不能与其客户端共享同一个地址空间。这种分布性需要一种有力的机制在分布式系统中定位组件，并访问其服务。然而，组件的分布性对客户端来说应当是透明的。
- **组件和客户端间的异质性。** 分布式系统中的组件可以采用不同的编程方式和语言来实现，它们可能与客户端所使用的有所不同。然而，尽管存在这种异质性，分布式系统中的客户端和组件还是应当彼此无缝地配置起来，而不会以“硬编码”的方式暴露出彼此对对方实现细节的依赖。

许多模式可以用于应对上述挑战。其中一部分是领域相关的。例如，用于定义组件具体职责

和约定的模式。这些模式超出了我们的分布式计算模式语言的范畴，要包含所有与应用领域相关的模式将会使得本书内容过于膨胀。所以，我们建议读者阅读后面这些文献中有关的部分[Fow97][PLoPD1][PLoPD2][PLoPD3][PLoPD4][PLoPD5][Ris01]。本章仅关注有助于划分组件接口和解决组件分布式相关的几个关键问题的模式。模式语言中有11个这样的模式。

- ❑ **Explicit Interface (163)** 模式将组件使用 and 实现细节分离。客户端只依赖于组件接口定义的契约，而不依赖于组件的内部设计、实现规范、位置、同步机制和其他实现细节。
- ❑ **Extension Interface (165)** 模式允许组件导出多个接口，在开发人员扩展或修改组件功能时避免接口膨胀和破坏客户端代码。
- ❑ **Introspective Interface (166)** 模式提供了一个辅助接口，以支持客户端访问组件类型、功能、公开接口、内部接口、行为，以及运算状态信息。客户端或应用之外的工具可以使用这些信息来监控组件或控制组件的使用。
- ❑ **Dynamic Invocation Interface (动态调用接口) 模式 (167)** 提供了一个辅助接口，以允许客户端动态地调用组件方法。它支持在运行时组织调用而不是在声明时选择接口。
- ❑ **Business Delegate (业务代表) 模式 (170)** [ACM01] 对使用组件的客户端封装了与访问远程组件相关的基础设施的各方面，如查找、负载平衡和网络错误处理等。Business Delegate 保证了在分布式应用中调用组件时的位置透明性。
- ❑ **Proxy 模式 (169)** [POSA1][GoF95] 使得组件的客户端能够与 Proxy——而不是组件本身——进行透明的通信。Proxy 能达到多个目的，包括简化客户端编程、提高效率，以及防止未授权的访问。
- ❑ **Facade (外观) 模式 (171)** 为子系统的一套接口提供了一个统一的、高层次的接口，从而使得子系统更易于使用。
- ❑ **Combined Method (组合方法) 模式 (172)** [Hen00c] 将通常一起使用的方法放在一起，形成一个方法来确保正确性，并提高在多线程和分布式环境中的效率。
- ❑ **Iterator (迭代器) 模式 (173)** [GoF95] 提供了顺序访问聚合元素的一种方式，它的好处是不需要暴露组件的内部表现。
- ❑ **Enumeration Method 模式 (174)** [Beck97][her01c] 将聚合 (aggregate) 组件上的迭代 (对组件中每个元素执行一次操作) 封装成聚合上的一个方法。其目的是减少外部迭代方式多次对聚合中的元素进行独立访问的开销。
- ❑ **Batch Method (批处理方法) 模式 (175)** [Her01c] 将重复访问的聚合元素合并在一起以减少单个元素的多次访问开销。

Explicit Interface 模式描述了所有接口设计策略的基本思想：在为组件定义表达性强且直观的接口时，将接口与其实现严格而明确地分离开。Explicit Interface 解决了前面描述的划分接口面临的最基本的挑战：约定规范、简洁性、表达性、质量、松耦合，以及组件分布式。因此，分布式计算模式语言中的许多其他模式都应用了这个模式，如图12-1所示。

Extension Interface 进一步细化了 Explicit Interface，它为组件引入了角色专属的接口，从而使得客户端可以从角色的角度去看待组件（见图12-2）。

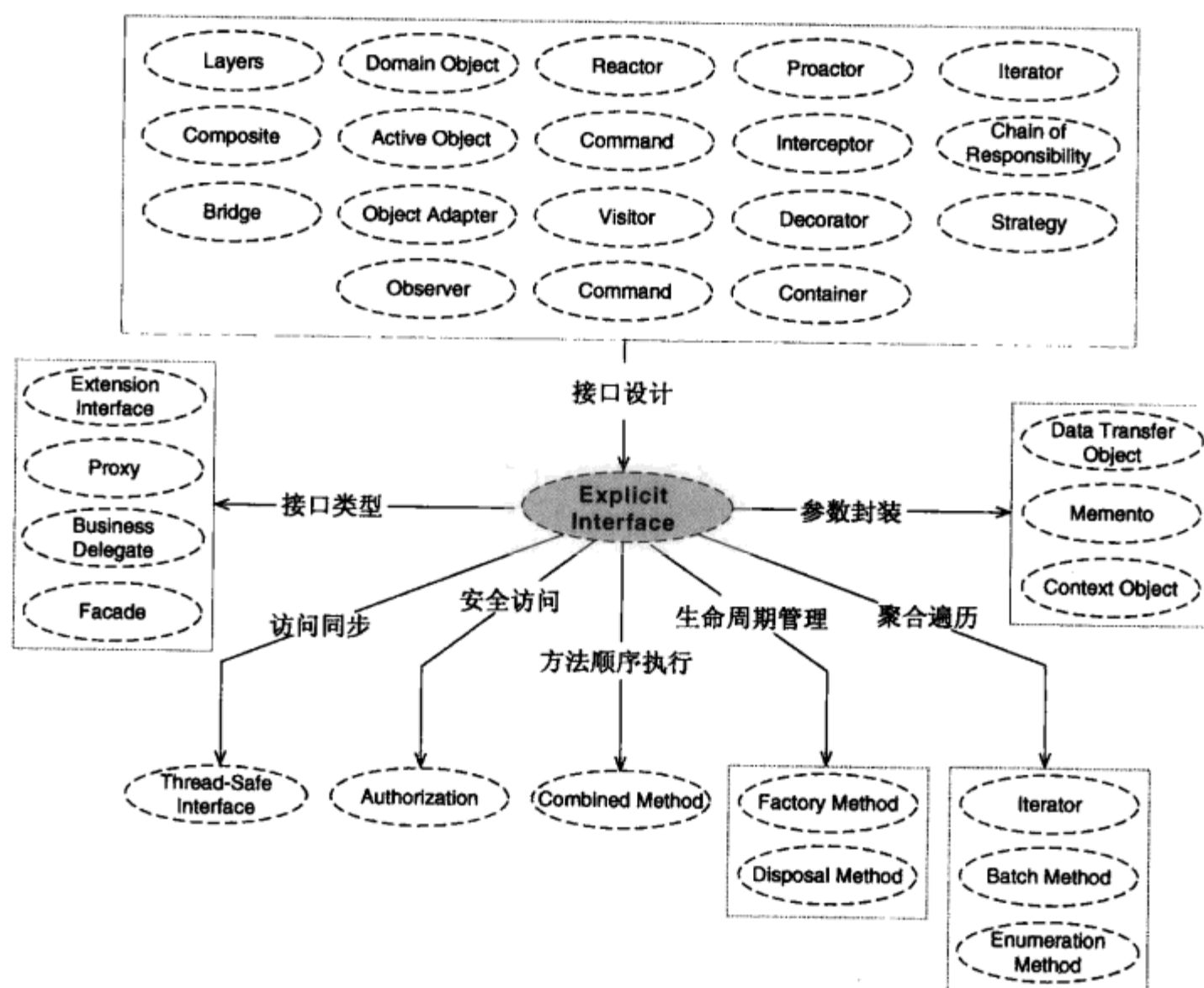


图 12-1

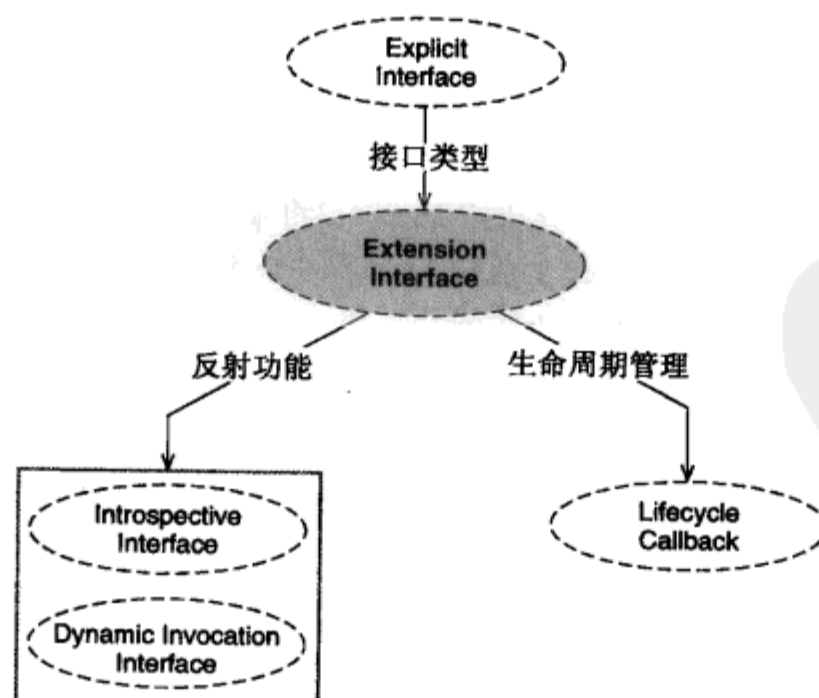


图 12-2

角色专属接口 (Role-specific Interface) 的概念有助于解决接口设计中的各种不同挑战和问题。

- 稳定性。客户端用不到的角色专属接口发生变化时，不会影响到客户端。
- 版本管理。如果对某个角色专属接口的方法进行升级，如修改、增加或删除方法原型，可以将其实现为单独的角色专属接口，其旧版本仍然得到支持。
- 扩展性。如果组件扩展了一个新的角色，我们可以把这个角色作为一个单独的接口提供给客户端，这样对新角色不感兴趣的客户端不会受到影响。
- 特定用途使用。在某些高级应用场景中，往往要求组件具有反射能力，而这并不一定是组件核心契约的一部分。专门的反射接口有助于将组件的“标准”使用和“特定目的”的使用区分开，以便“标准”客户端能采用显式的、类型安全的方式来使用它。类似地，单独的生命周期管理接口可以使得组件中间件（如容器）对组件生命周期的管理不会对客户端产生影响。

接下来的两个模式，Introspective Interface和Dynamic Invocation Interface为组件定义了“高级”接口，以允许客户端获取组件内部细节信息，而且客户端可以通过动态地组织请求的方式来调用组件的功能。这些反射性接口对于外部工具特别有用，如测试框架、系统监视器和调试器等，它们需要监视、控制和访问组件，而不依赖于特定的组件细节，包括其功能接口。

图12-3显示了Introspective Interface和Dynamic Invocation Interface是如何集成到我们的分布式计算模式语言中的。

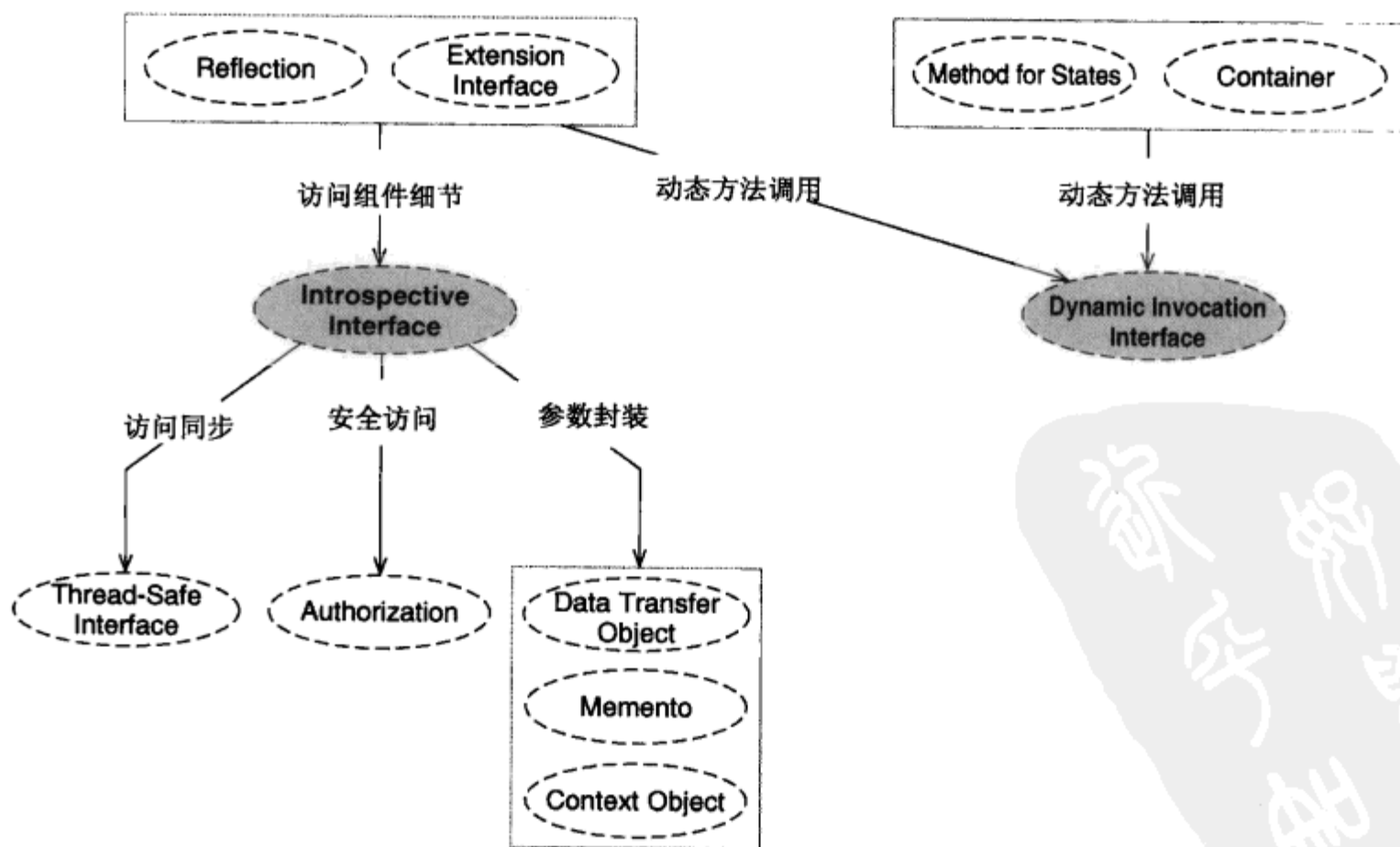


图 12-3

Business Delegate、Proxy和Facade这3种模式代表组件接口3种不同却又相关的设计风格。

- Business Delegate有助于将那些与分布式相关的基础设施方面隐藏起来，如负载平衡和复制。
- Proxy为不能直接访问的组件提供一个Proxy，这些组件有的是远程的，有的是保存在数据库中的，有的则是要求安全访问的。
- Facade为一组向用户提供各种服务的组件定义了一个访问点。
- 图12-4描绘了如何将Business Delegate、Proxy和Facade集成到我们的模式语言中。

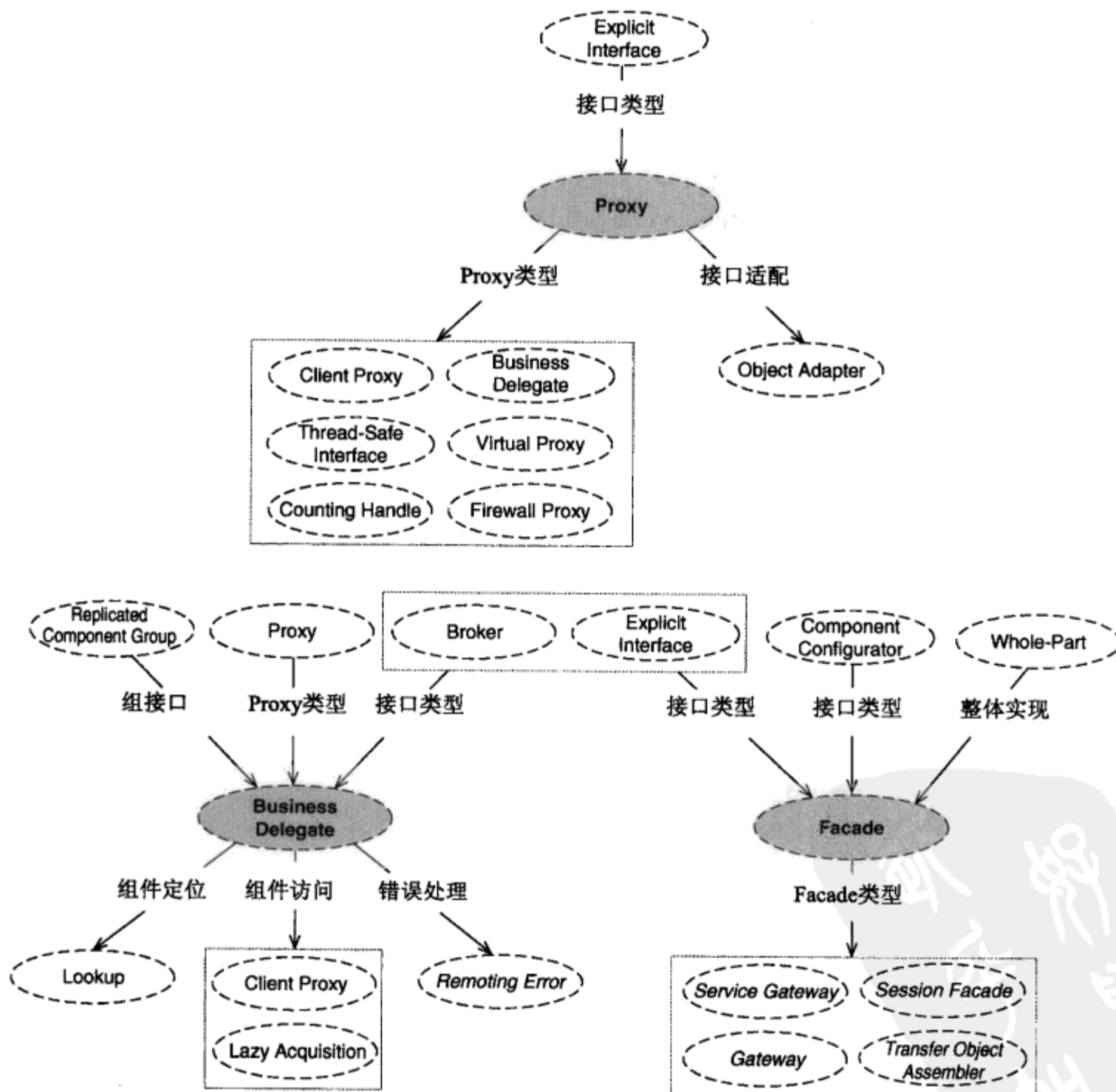


图 12-4

剩下的4个模式：Combined Method、Iterator、Batch Method和Enumeration Method用于设计

组件接口中的具体方法，并主要应用于构建并发性和分布式系统。图12-5展示了它们是如何集成到我们的分布式计算模式语言中的。

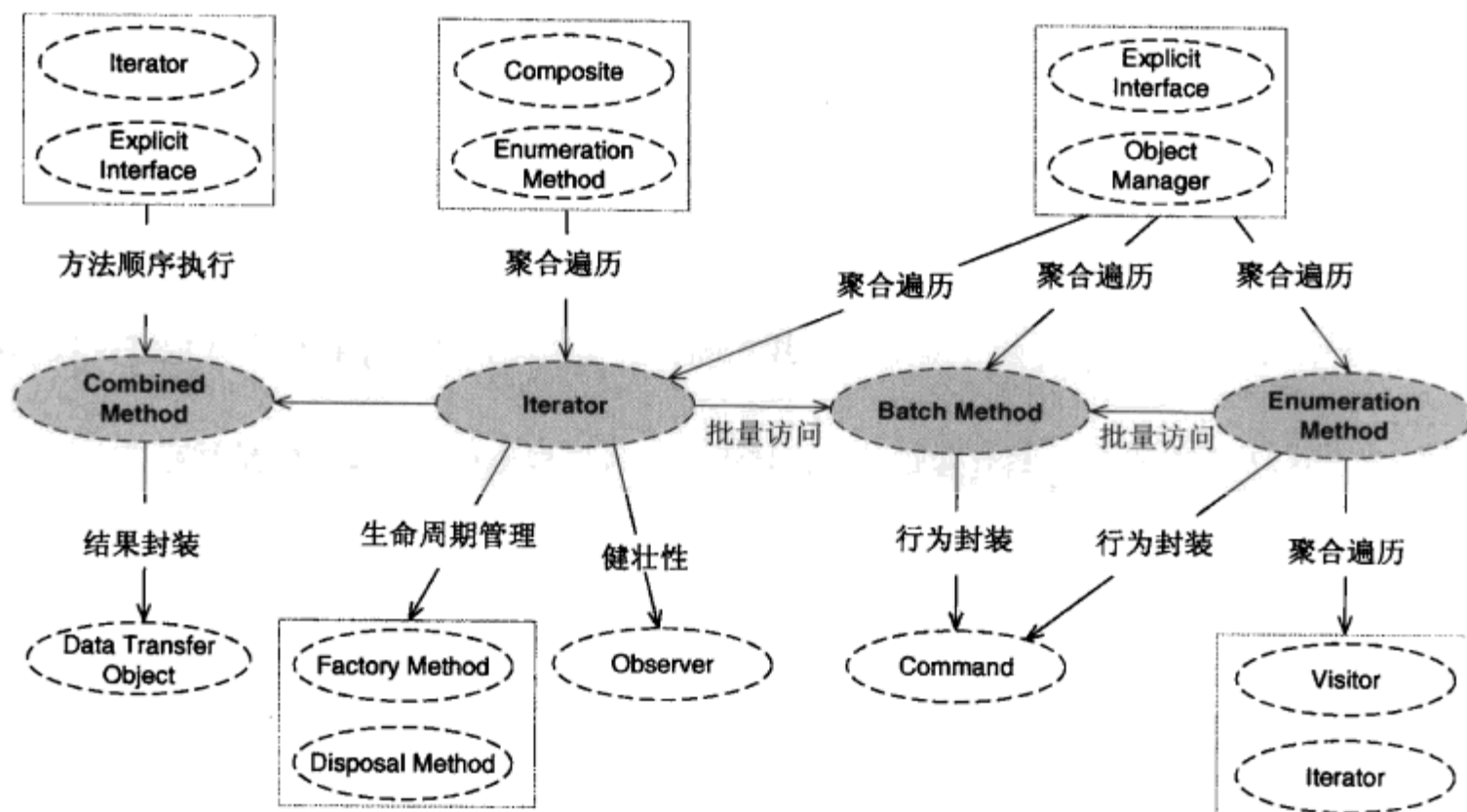


图 12-5

本章中展现的某些模式在其他文献中另有不同的几种形式。例如，Explicit Interface在*Server Component Patterns* [VSW02]中称为Component Interface，该书同时还包括Component Proxy模式——一种特殊形式的Proxy。应用于远程组件的，包含Combined Methods的Explicit Interface在*Patterns of Enterprise Application Architecture* [FOW03a]中被称为Remote Facade。对于Facade模式来说有很多特殊化应用。*Enterprise Solution Patterns using Microsoft .NET*中提出了Service Gateway模式，其实是为了一套远程组件提供的客户端Facade[MS03]。类似地，*Core J2EE Patterns*中提出了Session Facade模式，它是为了一套EJB组件提供的服务端Facade[ACM01]。*Core J2EE Patterns*还提出了Transfer Object Assembler，实际上也是用来聚合几个业务组件结果的Facade。*Patterns of Enterprise Application Architecture*中的Gateway模式也是一个针对外部系统的Facade。

12.1 Explicit Interface**

在设计Layers (108)、Domain Object (121)、Reactor (150)、Proactor (152)、Iterator (173)、Composite (185)、Active Object (212)、Command (240)、Interceptor (260)、Chain of Responsibility (257)、Bridge (255)、Object Adapter (256)、Visitor (261)、Decorator (262)、Strategy (266)、Wrapper Facade (269)、Observer (237) 或Container (288) 的时候……软件架构的所有工作主要是围绕着如何有效而恰当地表现组件接口。



组件代表着一个自我完备的单元，包括其功能、部署和使用协议。客户端使用这些组件来构建自己的功能。然而，直接访问组件的全部实现，会使得客户端依赖于组件的内部细节，最终会增加应用的内部软件耦合。

理想情况下，客户端应当只依赖于组件的公开接口。如果该接口保持稳定，则修改组件的实现不会影响到客户端。将组件封装在普通的具体类中是不明智的：这些类的接口总是与其实现绑定在一起。即使用抽象类，由于我们通常都会在里面包含一部分实现，这种绑定对于松耦合和稳定性也是不利的。位置无关性是另一个要考虑的因素：组件的客户端可能驻留在远程地址空间中，而且组件的位置可能在运行时改变，因此客户端应当避免依赖于组件的位置。最后，组件提供的方法应当对客户端是有意义的，并支持对其正确有效的使用，特别是在分布式或并发性的部署场合。

因此，将组件的声明接口与其实现分开。将组件的接口导出给客户端，但是对客户端保持其实现的私有性和位置无关性（见图12-6）。

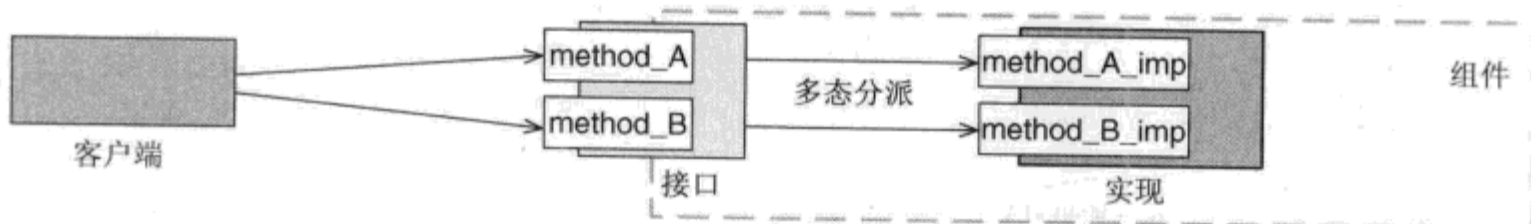


图 12-6

客户端对Explicit Interface的调用会被转发给组件，但是客户端代码应当只依赖于组件的接口，而不是实现。每个Explicit Interface都对应一个契约（contract）[Mey97]，客户端必须遵循该契约以便正确地使用组件。该契约包括组件提供的操作、调用操作的协议，以及客户端为了能够正确有效地使用组件而必须知道的其他约束和信息。



Explicit Interface在组件接口和具体实现之间进行了严格的划分，将组件的使用问题与具体实现及位置细节分离开。这种分离还使得我们可以独立于使用该组件的客户端，对组件的实现做透明的修改，前提是只要接口所定义的契约保持稳定。

下面几种模式有助于形成组件的Explicit Interface。Extension Interface (165) 支持将Explicit Interface划分为多个更小的接口，分别对应于组件的一个角色。Extension Interface还使我们能够为组件扩展新的角色专属的接口。简单地说，Extension Interface在支持接口改进的同时，降低了对组件客户端的影响。

对于与调用组件相关的内务操作（house-keeping task），我们可以用Proxy (169) 进行封装。例如，它可以将方法调用转换成可以通过网络发送给组件实现的消息；或者在第一次访问的时候从数据库中载入对象；或者为了提高客户端的访问效率将不变状态（immutable state）缓存起来。Business Delegate (170) 则更适用于动态分布式环境，这种情况下访问组件通常要求完成大量的基础设施任务。例如，在调用组件之前，必须找到其实现，并与其实现建立连接。Business Delegate可以在客户端调用组件方法时，以透明的方式为客户端执行这些任务。如果组件内部包含很多更

小的部件，我们可以使用Facade将客户端与组件的内部结构隔离开。Facade可以为组件提供单一的、定义好的入口点，从而使得组件结构改变时不影响客户端。

指定Explicit Interface的一个关键问题是操作质量（operational quality）：客户端必须能正确有效地使用组件。将Explicit Interface设计成Thread-Safe Interface (224)，可以将在并发使用环境中对组件的访问进行串行化，而且保持锁定开销最小；同时，在使用非递归锁（non-recursive lock）的情况下，避免在组件调用自身方法时出现自死锁。Authorization (204) 确保对组件功能的安全访问。Combined Method (172) 表示组件上的一系列方法总是按某个顺序一起调用，它使得Explicit Interface具有更好的表达性，因为它反映了组件的常见用途。Factory Method (313) 和Disposal Method (314) 允许客户端创建和销毁组件，而不需要依赖于其内部结构或者构造/销毁的过程。

如果接口表示的是一个聚合结构，比如集合（collection），则客户端可能需要访问其元素，或希望在其元素上执行操作。Iterator (173) 允许客户端挨个遍历这些元素而不破坏组件的封装性。Batch Method (175) 和Iterator类似，但是它更适用于分布式和并发性系统，因为它每次调用都会发送和返回多个元素，这有利于降低网络和同步的开销。Enumeration Method (174) 有助于在每个元素上执行特定操作，而不需要调用者显式地管理遍历的过程，从而降低了在并发性使用场景中的同步开销。

在Explicit Interface上的调用参数和结果可以封装成Data Transfer Object (244)，以避免组件依赖于具体的数据表现形式。如果客户端需要访问组件的内部状态，可以通过返回Memento (242) 的方式来保证封装性。如果组件需要客户端相关的信息执行其服务，可以将其作为Context Object 传给组件。

12.2 Extension Interface**

在实现Explicit Interface (163) 的时候……我们希望在接口升级的时候，确保客户端的稳定性和类型安全性。



只有在组件提供稳定而内聚的接口时，客户端才能有效地使用它。然而，功能的改变或者扩展往往会影响到组件的接口，这将会对客户端代码产生破坏——即使新功能并未使用。

理想情况下，当客户端没有使用的组件接口发生变化时，或者组件增加了客户端不关注的新服务时，客户端应当不受到影响。即使客户端确实使用了发生变化的接口，只要没有使用到变化的部分，它也不应当受到影响。同样，在组件的实现扩展新的服务或者组件更新已有服务的签名的时候，已存在的接口应当保持稳定。

因此，让客户端只通过特化的（specialized）Extension Interface访问组件，并且为组件提供的每个角色引入一个这样的接口。当组件需要增加新功能或更新已有Extension Interface的签名的时候，都要引入新的Extension Interface（见图12-7）。

关注组件特定角色的客户端通过相应的Extension Interface发出请求，Extension Interface再将请求转发给对应的组件实现。在实现组件的Extension Interface时，首先确定它在其全部预想的使用场景中可能扮演的各种角色。将每个角色封装到单独的Extension Interface，并只允许客户端通

过访问这些Extension Interface来完成各自的工作。避免修改已有的Extension Interface；在为组件扩展新功能或更新已有服务的签名时，创建新的Extension Interface。

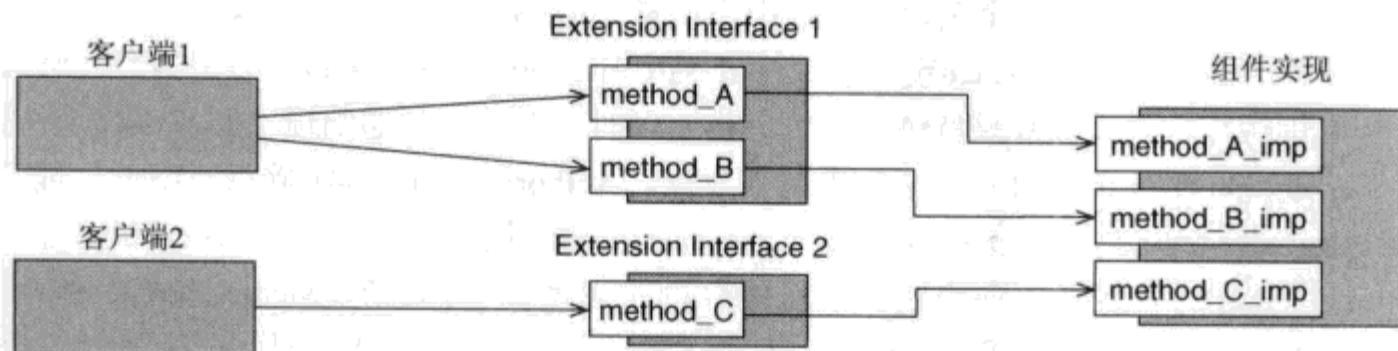


图 12-7



Extension Interface设计提供了以下几个好处。首先，它使得客户端与组件的耦合最小，客户端只依赖于它们实际使用的角色接口，这确保它们不会在组件增加新服务或方法签名改变时被破坏。其次，客户端仍然可以通过精确的强类型接口访问组件功能，而不必访问膨胀的、“万金油”式的接口或低效的、动态类型的、面向消息的接口。在Extension Interface设计里面，每个角色专属的Extension Interface可以是一个Explicit Interface，在这个Explicit Interface中包含——并且只包含——需要完成各自角色的方法。

为了管理组件的Extension Interface，引入一个专门的根接口用来获取和访问组件的每个Extension Interface。另外，根接口还可以提供了以下（可选的）功能：Introspective Interface (166)，它允许客户端获取组件信息；Dynamic Invocation Interface (167)，客户端可以通过它向组件发出请求，而不需要使用任何角色专属的Extension Interface。当组件被外部工具监视或访问时，这两种接口尤其有用。比如，有时候我们需要将其放入测试框架中或置于系统监控器的监控之下，或者需要动态地集成到原先并不是为该组件设计的应用中。

Lifecycle Callback (295) 和配置功能是根接口的可选功能，应用可以用它们来初始化组件并主动控制其生命周期。必须确保根接口功能可以通过组件的任何一个Extension Interface访问，例如通过让所有Extension Interface继承自根接口，或在所有Extension Interface中实现其功能。

12.3 Introspective Interface**

在实现Reflection (114) 架构或Extension Interface (165) 的根接口的时候……有时我们必须允许客户端访问它们所使用组件的信息，即元数据。



客户端要正确地使用组件可能需要访问组件的某些信息，例如其类型、标识、所支持接口或当前状态。然而，允许客户端直接访问这种技术细节可能会破坏组件的封装，并降低依赖的稳定性。

此外，客户端会依赖于组件的结构，这将提高应用的复杂性，并使得维护和升级更加困难。

for States (275) 结构或者Container (288) 的时候……我们可能需要允许访问组件的功能而不必知道或使用其静态类型接口。



组件的Explicit Interface声明并提供了一系列协议，但有时候我们需要组件支持额外的方法调用。当客户端必须调用组件事先不知道的额外功能时，这种开放性就显得非常必要了。

组件可能被动态载入到框架环境中，例如客户端的用户界面组件、服务端的业务逻辑组件，或者自动测试工具中的一组测试用例。载入的组件能支持与自身任务有关的各种方法，但这些方法通常是跟调用框架无关的——除非框架必须知道组件有哪些方法，并能够向组件转发事件和传递调用。

通常，我们无法限制组件所可能包含的特性，比如组件可能需要大量的测试用例来验证其特性。另一方面，这些特性可能形成一个内聚的模型，但并不存在一个通用的框架专门处理该模型。例如，用户界面控件可以选择只支持GUI框架接受的几个事件和属性。Explicit Interface (163) 可以捕获其提供的特定细节，但不一定是这种动态客户端能使用的方式。Introspective Interface支持方法查询，但并不一定允许动态客户端调用。

因此，为那些允许客户端动态调用的组件引入一个Invocation Interface。在运行时通过字符串来定位方法，而参数则通过类型化的集合的方式传递。将Dynamic Invocation Interface与组件的“操作”接口隔离开（见图12-9）。

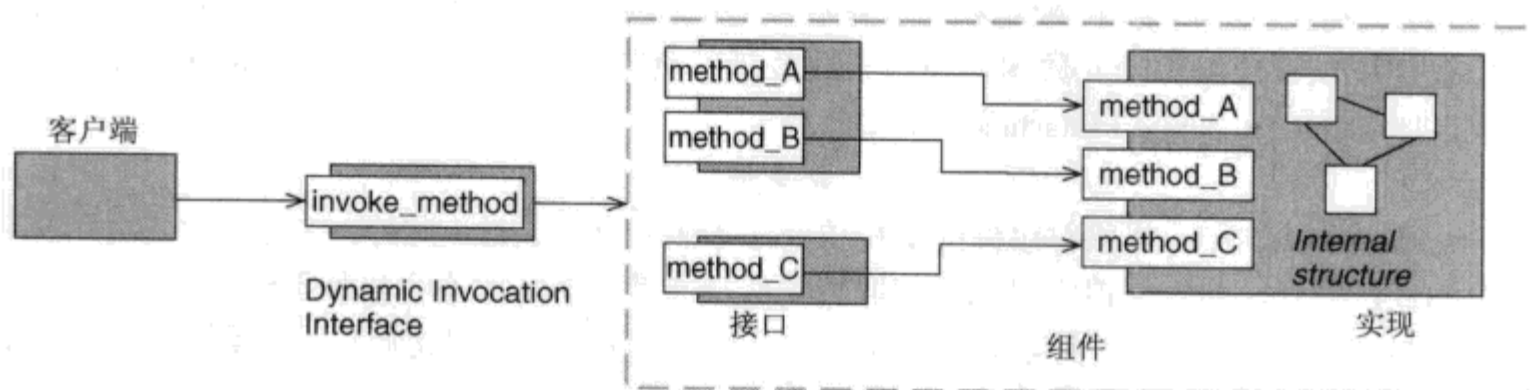


图 12-9

通过Dynamic Invocation Interface，对组件的调用会被分发给组件声明接口中的特定方法。



Dynamic Invocation Interface为客户端调用组件提供了另一种选项。这种方式基于动态协议，而不是静态的Explicit Interface，所以是开放的、非侵入的，它增强了客户端应用的灵活性。

然而，这些好处是有代价的，正如任何形式的延迟绑定技术都会带来相应的不足：对错误实现或错误的协议用法不能及时地做出检查。让组件遵循统一的命名或标记方法的协议非常重要，但这仍然不能确保正确性。可以采用某种检查工具来检测对动态接口的错误调用，但使用这种方式比静态检测调用有更多限制。类似地，对动态调用的自动重构也不像对静态检查接口的自动重构那样能保证正确性。

Dynamic Invocation Interface的可改进性和灵活性可能也会和性能开销有关。在具有静态类型

检查系统中，大量使用基于反射的检查 and 调用相对于执行已声明接口要昂贵得多。这种性能开销在解释性语言中可能不是太大的问题，如Smalltalk、Ruby和Python，它们的类型系统和正常执行模型本身就是基于动态模型之上的。

12.5 Proxy**

在实现Explicit Interface (163) 的时候……我们经常希望避免直接访问组件实现的服务。



软件系统中包含的组件是互相协作的：客户端组件访问并使用其他组件提供的服务。通常，直接访问组件的服务是不现实的，甚至是不可能的，例如我们必须首先检查客户端的访问权限，或者是因为组件的实现位于远程服务器中。

由于以下两个原因，我们通常不希望在组件中包含“内务操作”功能，如授权等。首先，我们可能不需要在组件的每次使用中访问这些功能。其次，这样会导致多个正交的方面混合在单个实现中，进而使得其难以单独修改每个方面。组件的功能应当和所有的内务操作相独立。基于同样的原因，我们并不希望客户端来完成这些操作，因为这样会导致客户端和组件实现紧耦合。例如，如果客户端去直接访问远程组件，它们将会依赖于组件的位置以及用来访问其功能的网络协议，而这对组件的客户端应当是透明的。

因此，将所有组件的内务相关功能封装成一个单独的组件替代者——Proxy，并让客户端只通过Proxy通信，而不是直接访问组件本身（见图12-10）。

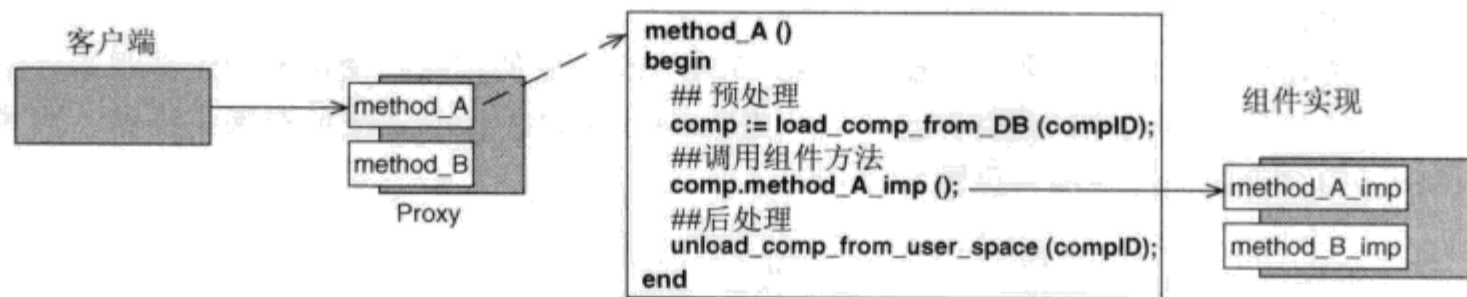


图 12-10

设计Proxy时要让它提供与组件相同的公共接口。当客户端调用Proxy的方法时，首先做一些必要的内部预处理工作，然后再将客户端请求转发给“真正的”组件，由Proxy调用组件的相应方法。当控制流返回时，在将方法的结果返回给客户端前，完成所有必需的后处理操作。



Proxy方式使得客户端和组件都不需要实现组件相关的内务功能。而且对客户端来说，连接到组件本身还是其Proxy是透明的，因为二者的公共接口是相同的。Proxy的主要缺点是它隐藏了给客户端带来的开销，尽管很多情况下，这种开销和组件服务的执行事件相比是可以忽略的。

Proxy类型[POSA1]存在多种。Client Proxy (139) 使远程组件的客户端不必关心网络地址和IPC协议，从而在分布式系统中提供位置无关性，客户端能像使用本地组件一样来使用客户端Proxy。Business Delegate (170) 则更进一步：在分布式应用有多个可用组件的情形下，它为客户端屏蔽了所有IPC、远程组件定位、负载均衡工作以及处理特定网络错误。Threadsafe Interface (224)

是一种将组件并发访问串行化的Proxy，对客户端和组件都是透明的。Counting Handle (309) 通常也表现为Proxy，帮助我们访问共享堆对象的功能，其生命周期必须由应用来显式管理以避免对象不用时产生内存泄漏。Virtual Proxy (294) 在需要的时候载入或创建昂贵的组件并在使用完后从内存中删除它。最后，Firewall Proxy (202) 保护软件系统免受特定类型的外部攻击。

如果为Proxy和它所代表组件提供相同的接口是不可能或不现实的，则可以利用Object Adapter (256) 来映射这两个接口。

12.6 Business Delegate**

在实现基于Broker(137)的分布式基础设施、Explicit Interface (163)、Proxy (169) 或Replicated Component Group (189) 的时候……我们经常必须考虑组件的访问来源可能来自于另一个地址空间。



因为网络的性能和可靠性等原因，访问远程组件和访问本地组件有很大的不同。理想情况下，客户端不需要关心其使用的组件是本地还是远程的。

跨越网络的访问包括特定的基础任务，如获取远程组件的位置、错误处理、负载平衡等。许多这样的任务对本地组件来说是不必要的。如果客户端依赖于组件的特定位置，或者甚至依赖其远程性，它们将不得不自己完成这些任务，这将会使得处理网络代码和领域相关代码混杂在一起，增加应用的复杂度。此外，当它们使用的组件由于错误恢复或者负载均衡等原因导致位置发生变化时，客户端也需要做相应修改。因此，客户端代码应当尽可能地独立于所调用组件的位置及管线(plumbing)支持。

因此，为每个远程组件引入一个Business Delegate，它的创建、使用和销毁都跟本地组件一样，而且其接口与所代表的组件完全一致。由Business Delegate完成所有网络任务，这对使用组件的客户端来说是透明的（见图12-11）。

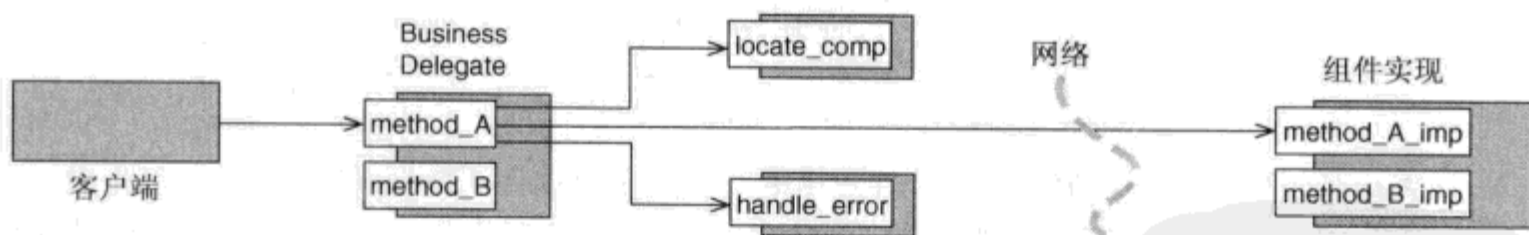


图 12-11

当客户端创建和访问Business Delegate时，Business Delegate会找到其代表的远程组件。对Business Delegate的后续方法调用可根据位置信息转发给远程组件。此外，Business Delegate还负责与远程组件通信时的错误处理。如果部署了组件的多个实例，Business Delegate还可以在向具体组件实例发送请求之前完成负载均衡功能。Business Delegate还是系统管理功能的理想接入点，以监控所有客户端和远程组件的通信和交互。



在分布式系统中，Business Delegates支持位置无关的组件调用，使得网络任务和事件对

Business Delegate的客户端是不可见的。此外，每个Business Delegate与客户端本地组件共享同样的生命周期和使用协议，它的创建和销毁可以在构造函数、析构函数或垃圾回收器中实现，其调用也可以通过“普通”的方法来实现。Business Delegate的主要不足在于可能为客户端引入的隐蔽开销，尽管在大多数使用场景中，其开销和Business Delegate主要职责的执行时间相比几乎可以忽略不计。

通常情况下，Business Delegate使用Lookup (292) 服务来获取所代表远程组件的一个或多个实例。对特定远程组件实例的访问经常封装在Client Proxy (139) 中，以便Business Delegate无需关心访问该实例的网络协议的细节。Lazy Acquisition (300) 可以帮助我们将与组件建立连接的时间推迟到第一次通过Business Delegate访问的时刻。

如果对远程组件的调用返回了Remoting Error [VKZ04]，Business Delegate可以先对错误进行适当的处理，然后再返回给客户端。例如，如果到组件的连接中断，Business Delegate可以试图重新建立连接并重发调用请求。相似地，如果某个组件实例负荷过重，Business Delegate可以尝试调用同一组件负荷较轻的实例。

12.7 Facade**

在开发Broker (137)、Explicit Interface (163)、Whole Part (183) 结构或者Component Configurator (289) 的时候……有时我们需要访问一个组件群，它们共同为客户端提供某些服务。



复杂的服务通常是由一个组件群提供的，其中每个组件都为客户端提供一个自我完备的服务。如果客户端想调用复杂的服务，它必须显式地维护与其中每一个组件之间的关系，然而，这样将会使它们依赖于该组件群的内部结构。

如果组结构发生变化，所有客户端都会受到影响。此外，这样的依赖越多，软件系统的物理和逻辑复杂度就越高[Lak95]。理想情况下，客户端应当通过单一入口来访问相关的组件群，这样从客户端的角度来说就简化了任务的调用和执行。另一方面，也有一些客户端只使用群中的特定组件。强制这些组件通过单独的组件入口访问该组件的方法会引入不必要的间接性，进而导致性能上的损失。此外，还有的客户端可能希望在组件群上执行某些更加复杂的任务，这要求组件的集成方式和调用方式都会与通常的情形有所不同。

因此，为组件群指定单一的访问点——Facade，在通常的使用场景中将客户端请求中转给合适的组件，同时也允许在特定的更复杂情况下绕过该访问点（见图12-12）。

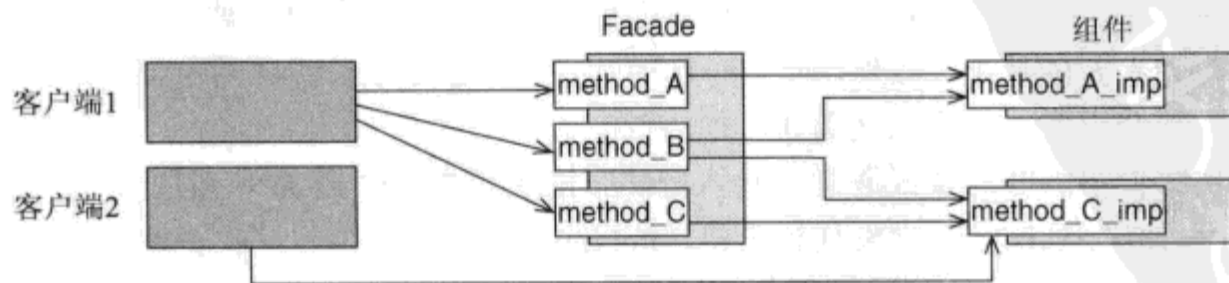


图 12-12

除了转发请求，Facade接口在其公开接口和“被保护”的组件接口间完成所有必需的适配工作。Facade还可以将不同组件的特性聚合成新的、“更高级”的服务。然而，客户端并不是必须通过Facade来调用组件，客户端仍然可以直接访问特定组件。



通过Facade调用组件群的客户端相对于组件群的内部结构和内部关系是独立的，因此当结构发生变化时它们不会受到影响。此外，想要访问组件群中特定组件的客户端也可以绕过Facade直接调用，从而避免这类客户端性能上的损失。

在远程应用设置中，Facade可以部署成客户端地址空间的Service Gateway[MS03]或者（服务端）组件群地址空间的Session Facade[ACM01]。Service Gateway有助于提高吞吐量和可伸缩性。在客户端地址空间内完成选择调用的组件将对Facade的请求适配成相应组件的接口。Service Gateway的缺点是组件群中的每个组件都是远程访问的，这会增加网络开销以及分布式应用的复杂性。

Session Facade避免了上述缺陷——远程客户端只能通过Facade访问组件群。然而，客户端访问组件越频繁，就引入越多的转发和适配开销以便将请求转发到组件群中的相应组件上，这时，Session Facade就越容易成为性能和伸缩性的瓶颈。

Gateway [FOW03a] 是另一种形式的Facade，它代表了应用所使用的外部系统的访问点。因此，应用独立于外部系统的特定接口及其内部结构。Transfer Object Assembler[ACM01]是将从几个组件接收到的结果组合在一起的Facade，它可将结果组合在一起返回给客户端。

12.8 Combined Method**

在实现Explicit Interface (163) 或Iterator (173) 的时候……我们经常需要按相同的顺序来调用某个组件的多个方法。



客户端经常必须按照同样的顺序调用组件的多个方法以执行特定任务。然而，从客户端角度来看，每次都显式地顺序调用它想执行的这些组件方法既繁琐又容易出错。

在整个应用的方法多次重复相同的调用序列会增加开发、理解和维护的难度。在调用时弄错顺序、忘记调用某个特定方法或传入错误的参数是很容易发生的。此外，改变调用顺序，或者修改被调函数的签名会影响所有执行该方法序列的代码。

在分布式和并发性系统中还会有更多的问题：每次远程调用都会引入网络开销；而且对于可变的组件来说，即使每个方法对于竞争状态都是安全的，由于组件可能被多个线程共享，调用序列不一定能得到期望的结果。此外，如果序列中的任一方法出错，都必须由调用的客户端负责处理错误并撤销或回退在组件出错之前执行的方法。

因此，将组件中那些必须或通常一起执行的方法组合成一个方法（见图12-13）。

希望执行方法序列的客户端发送请求给Combined Method，由它来代替客户端执行该序列，此外，Combined Method的实现处理了与调用方法序列相关的所有分布性、并发性和故障管理方面的任务。

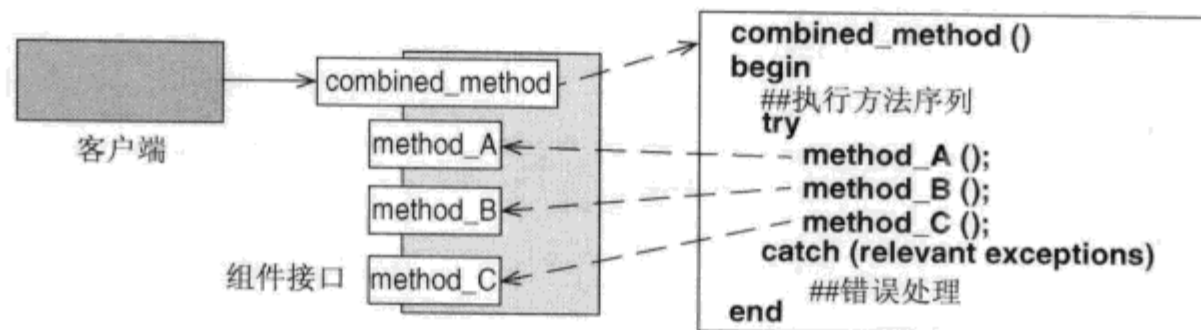


图 12-13

使用Combined Method增强了组件或对象接口的表达能力和内聚性，因为它反映了通常用法。类似地，应用的健壮性也得到了增强，因为只有一个地方，而不是很多地方都来编写方法序列并处理错误。因此，从客户端角度来说，组件或对象更易于使用。

分布式开销同样对Combined Method只发生一次，因为它将多个远程调用组合成了一个调用。类似地，非确定性导致的并发性故障被消除了，因为整个调用序列是同步的，而不是单独地调用方法。最后，错误处理和恢复策略都封装在Combined Method中，形成了更加类似事务风格的方法设计：要么整个调用序列都成功执行（对组件或对象产生相应影响），要么序列中所有方法都失败（对用户来说相当于整个序列根本没有执行，即保持组件和对象不变）。

用于查询或设置一组值的Combined Method可以表现为Data Transfer Object (244)。如果这种访问遍历聚合（Aggregate），如集合（Collection）中的每个元素，则Batch Method(175)可以认为是Combined Method的一般化表现形式——它不是将对象上的不同方法组合起来，而是把对目标对象的连续元素的访问合并到一次调用中。

12.9 Iterator**

在实现Explicit Interface (163)、Enumeration Method (174)、Composite (185) 或Object Manager (291) 的时候……我们经常需要顺序访问聚合的元素而不暴露其内部结构。

客户端经常希望遍历封装在聚合中的元素，例如集合中维护的元素。然而，客户端在访问自己感兴趣地组件时，可能并不希望依赖于聚合的内部结构，同时聚合也不希望将其内部结构暴露给客户端。

更为复杂的是，客户端经常需要按最适合它们需要的特定顺序遍历组件。多个客户端可能还希望同时访问聚合，甚至一个多线程客户端有可能同时对聚合进行多个遍历。然而，在聚合中直接支持多种遍历策略和并发遍历，会使其内部结构复杂化。例如，聚合必须在单独的会话中维护每个活动（active）遍历的具体状态。这也使得开发人员不能专注于实现聚合的领域职责。

因此，将聚合维护的访问和遍历组件的策略做对象化，形成一个单独的Iterator组件。让Iterator成为客户端访问组件的唯一方式，并允许Iterator访问聚合必要的代表以便其完成遍历（见图12-14）。

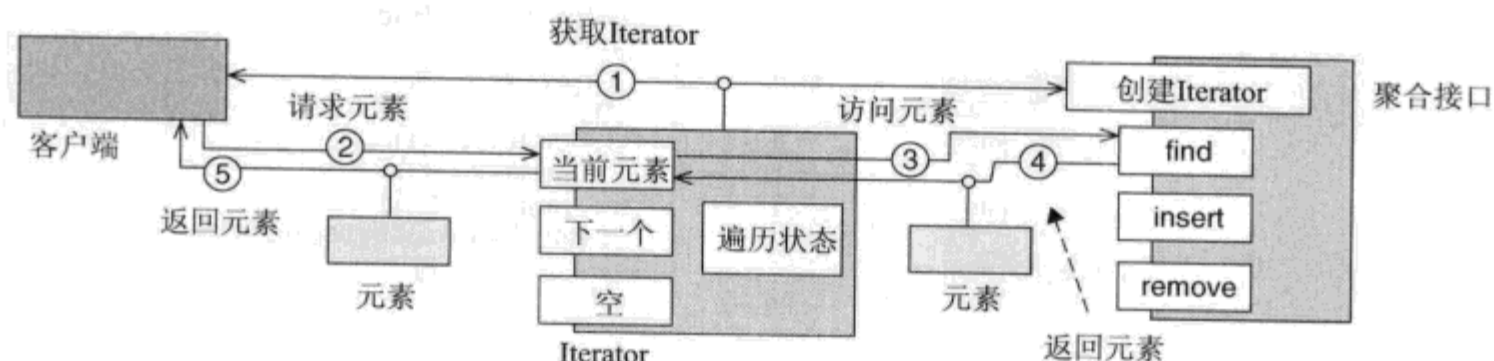


图 12-14

希望遍历聚合中元素的客户端必须首先从聚合中获取Iterator。然后客户端可以使用该Iterator按照一定的顺序访问这些元素。



Iterator在保留聚合的封装性的同时保持了客户端与其内部结构的独立性。此外，将访问和遍历策略具体化以允许多个客户端维护各自在聚合内容上的遍历状态，或者单个客户端同时运行几个独立的遍历。不过Iterator方式对单进程、单线程组件部署方式最适合。

Iterator的设计通常是基于抽象的Iterator的，该抽象Iterator实现了Explicit Interface，以访问和遍历聚合维护的组件。具体Iterator继承或实现该接口以实现具体的访问和遍历策略，如树状结构的广度优先或深度优先策略。这种设计有助于将不同遍历策略封装在统一的接口中，以及集成新的或改进已有的Iterator类型而不需要修改聚合中已有的迭代器管理基础设施。

在Iterator接口上运用Combined Method (172) 避免了运行在不同线程的Iterator访问聚合时出现难以发现的竞争状态。类似地，在Iterator接口上运用Batch Method (175) 能支持对聚合中的元素做“大块”（chunky）访问，这样可以降低在访问远程聚合时带来的性能损失和无谓的网络负荷。

为聚合的接口提供Factory Method (313) 及Disposal Method (314) 以根据客户端请求创建和销毁具体的Iterator。这两个方法将Iterator的生命周期控制分离并封装到公开的接口中，并将其与聚合的领域逻辑隔离开。如果聚合的内部结构可以在遍历时修改，那么可能需要更健壮的Iterator。健壮性可以通过Observer (237) 方式得到：聚合扮演Subject的角色，并在其内部结构变化（例如删除聚合中的某个组件）时通知所有活动的Iterator [Kof04]。

12.10 Enumeration Method**

在Explicit Interface (163)、Batch Method (175) 和Object Manager (291) 中……我们可能希望对聚合中的每个元素逐个调用其某个行为。



某些类型的聚合，如图或树，其表现不适合使用基于Iterator的方式来进行遍历。类似地，使用Iterator来访问线程间共享的聚合中的元素时，反复的锁操作会引入不必要的开销。对远程聚合访问所引入的开销更大。虽然如此，我们还是需要高效访问聚合的元素以执行相应的操作。

更加糟糕的是，很多时候当聚合需要在遍历之前或遍历之后执行预处理或后处理代码。最明

显和常见的例子是线程中断的同步（Synchronization against threaded interruption）。由聚合的客户端自己编写这些代码既繁琐又容易出错。例如，分布式系统中的Java线程同步，外部同步块很容易出错，因为它会给你一个虚无的安全错觉[Hen01c]。

因此，将迭代放在聚合内部并封装成一个的Enumeration Method以负责完成遍历。将循环的任务——要在聚合的每个元素上执行的动作——作为参数传递给Enumeration Method，再应用到每个元素上（见图12-15）。

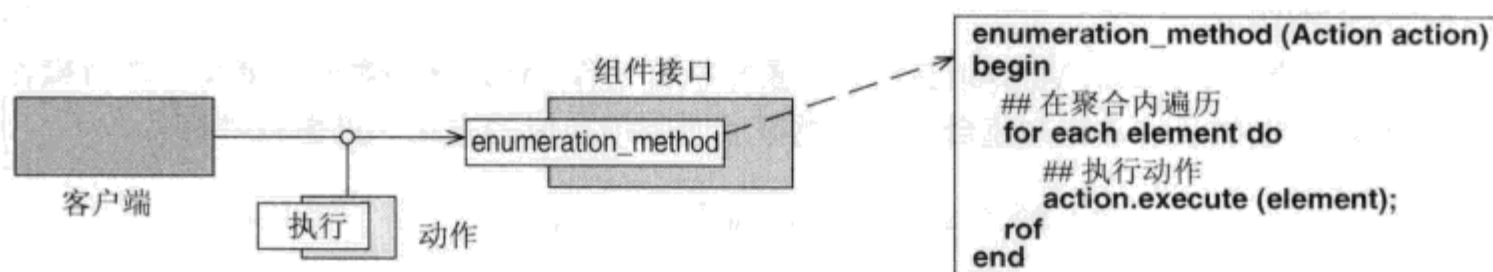


图 12-15

和Iterator方案不同，Enumeration Method执行完整的遍历，包括对所遍历聚合上的每个元素执行一个操作，是全部元素，而非多个独立的片段。



在分布式的网络环境中，将许多远程调用减少为单一的Enumeration Method能提高性能、减少网络错误以及节省宝贵的带宽。这些好处的关键在于Enumeration Method实现了控制反转的原则——不是客户端控制迭代，而是聚合自己来控制。此外，Enumeration Method也适用于对聚合本身在执行循环进行遍历之前或之后执行某些动作，例如同步。该方案允许聚合保留对其行为的控制：其设计更加完整、明确，封装性更好并且是自我完备的。

传递给Enumeration Method的行为是一个Command (240) 对象，或者某种类型的方法引用，例如C++中的函数指针或C#的Delegate。这样的设计为聚合提供一个“通用”的Enumeration Method，可以让客户端传递任意行为。然而，在远程访问时会存在一个问题。为了使控制反转保持高效，对Command的调用必须是本地而不是远程的。这就意味着Command对象必须从客户端复制到服务器，并且不应该作为远程对象来间接访问。这种限制也意味着Command的代码在被调用时刻必须是本地的——或者是已经存在本地，或者是在第一次调用时传递过来。这种代码传递意味着该模式不能应用在异质系统中。Enumeration Method在多线程情形下特别有效。Batch Method可能更适用于远程访问。

Visitor (261) 有助于Enumeration Method来简化非线性聚合结构（比如图）上的遍历，使其不依赖于具体的结构。反之，如果聚合的结构是线性的（如双向链表）Iterator (173) 就能实现这种独立性。

12.11 Batch Method**

在Explicit Interface (163)、Iterator (173) 和Object Manager (291) 中……我们可能希望在聚合上执行大批（bulk）访问。



有时候客户端希望在聚合上执行大批访问，例如从集合中获取满足某种特性的所有元素。如果访问聚合开销很大，比如它是远程的或并发性的，那么单独访问每个元素会引入相当大的性能损失和并发性开销。

如果聚合是远程的，每次访问会引入延迟和抖动，占用网络带宽，并引入额外的故障点。如果聚合是并发性组件，每次访问的开销中还得把同步和线程管理计算在内。类似地，每次调用还要完成一些内务性代码，如认证等，这进一步降低了性能。尽管如此，我们必须能有效且完整地执行大型访问。

因此，设计单独的Batch Method在聚合上重复执行操作。该方法的声明接受每次执行操作所需的全部参数，例如通过数组或集合；结果的返回也采用类似的方式（见图12-16）。

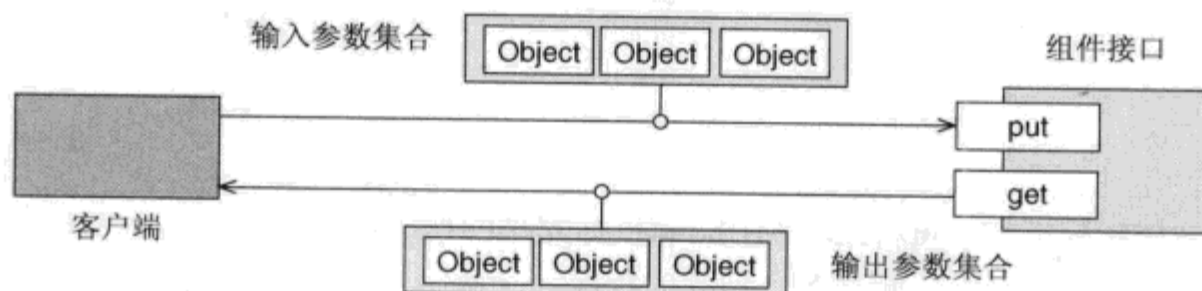


图 12-16

Batch Method将重复操作合并到数据结构中，而不是在客户端进行循环，因此循环是在调用前后执行的，分别为调用做准备及完成后续操作。



Batch Method将访问聚合的开销减少为一次访问或几次“大块”（chunked）地访问。在分布式系统中，这样能显著提高性能、减少网络错误以及节省宝贵的带宽。尽管使用Batch Method后对聚合的每次访问更加昂贵，但大批访问的整体开销降低了。大批访问也可以在方法调用内部进行适当的同步。

Batch Method在复杂性方面所作的妥协是它要完成大量内务操作以建立并处理调用结果，并要求更多中间数据结构传递参数并接收结果。然而，网络并发性的开销越高，其他每次调用要做的内务操作越多，相应的Batch Method的付出就越值得。Batch Method可以被视为Combined Method的一般化，它把整个循环遍历合并到一起，而不仅仅是短短的几个不同方法序列。

简单来说，Batch Method的参数要么是输入参数要么是输出参数，它们可以用简单的集合来表示，如值数组或键-值对的数组，或者甚至是参数变量列表——如果存在适当的类型安全的语言支持的话。输入参数列出了调用者要发送给聚合的元素，如要添加的元素或查询的关键字。输出参数可能由返回值来表示，列出聚合的返回结果，如找到的值。

通常Batch Method是特定目的而不是通用的。它们以特定操作命名而且其参数直接反映了输入和结果。例如，查找匹配键值的结果可以表示为单一方法。但是，如果需要更加通用，则需要更多的参数以控制所封装的循环。例如查找所有早于特定日期或大于特定值的条目。更加一般化的方式可能是传入一个判断或采用Command对象（240）形式的控制代码，这就使得Batch Method更像是Enumeration Method（174），它们在分布式环境中具有相似的不足。



Anna Buschmann正在拆乐高拼装玩具
©Frank Buschmann

组件是实现的构建块 (building block)，它为客户提供定义良好的服务。通常，客户端并不关心这些服务是如何实现的。组件的使用者当然可以得益于组件的黑盒特性，但是组件的开发者却不得不面对组件内部设计的各种挑战。因此，本章提供了6个模式帮助读者构建组件的实现。这些模式所关注的是组件的分解和对分布式系统各种组件部署情形的支持。

基于组件的软件开发是构建现代软件系统的关键技术。其核心的思想是，软件可以由定义良好的构建块组成，每个模块通过接口提供一个特定的、内聚的、完备的服务。组件及其可组合性是基于它们的强封装结构的，而且一般来说在绑定时间^①和绑定位置都有很大的选择空间。实现细节的强封装特性对基于组件的软件开发会带来两方面的好处。

□ 生产力。对于一个应用，如果它所需要的组件已经存在，或者可以通过第三方购买，这

^① binding time，有的地方译为“汇集时间”。——译者注

会显著地提高软件开发的生产效率。

- 质量。如果使用（或重用）的已有组件具有良好的口碑，软件的质量也会得到保证。因为这些组件已经经过测试、调试和调整，开发人员了解，并且可以依赖组件的属性。

在实践中要实现一个好的组件，对其内部实现细节进行良好的封装却并非易事。其原因如下。

- 组件划分。从外部来看，组件是一个完备而统一的构建块，它为其客户端提供定义良好的服务。然而，从组件的内部，或者从开发的角度来看，这种观点就行不通了。尤其是庞大的组件往往难以理解、改进和维护，其实现代码中到处充满了“意大利面条”式的结构。为了保证组件的可理解和可维护等内部质量，我们需要把组件划分为更细粒度的结构。然而要做出正确的划分，却也不是一件容易的事：组件的各个组成部分，以及它们之间的关系和交互应当能够恰当地反映组件的主要职责；同时，每一个特定的部分对于程序员来说又必须在语义上是有意义的，而且是可维护的。
- 组件质量。虽然客户端不大关心他们所使用的组件具体是怎么实现的，但是它们所调用的服务的功能行为和操作质量对它们来说却是非常重要的，这包括基本的功能、性能、可伸缩性、吞吐量等。要满足这些服务质量，并不是仅靠复杂的算法和优化的代码就能达到的，组件的内部设计对其影响也不可小觑。
- 组件灵活性。为了适应各种应用，组件必须为其相应的应用领域建立合适的模型。其内部结构必须支持各种设计方式，比如针对特定客户的需求的适应性、针对新特性的扩展性和针对新平台的可移植性。这就要求必须对组件的职能进行全面细致的考虑，比如在其应用情形中哪些是不变的，哪些是变化的。
- 对组件功能进行分布。根据定义，组件就是一个特定的逻辑或功能单元，或者是一个完备的实体[Szy02]。我们把这个定义放到分布式系统中，需要给每个组件安一个家：它位于某个特定的网络节点上。然而，这种整体的观点有时候却并不可行，尤其是（远程的）客户端在性能、可用性和伸缩性方面要求它所使用的组件提供较高的服务质量的时候，我们就会发现这种定义或者观点有其局限性。

这时候我们最好是将一个组件划分为一组更小的、专注于某个方面的组件，并且把这些小组件分布到不同的网络节点上。同样，我们也可以在系统中部署组件的多个实例。然而，对于其客户端来说，这些分布的组件组看起来应该是一些内聚的单元，在一个唯一的入口点上提供一套有意义的服务。这种做法要求通过适当的机制和协议显式地协调分布式组件组内部的协作。

- 组件实现内部的并发和并行。分布式系统和多核的处理器使得我们可以并行地执行组件实现内部的某一部分，这既可以提高单次调用的性能，同时也可以提高系统的可伸缩性和吞吐量。要利用网络或者计算机的并行能力，需要对组件实现进行适当的划分，将其分割成较小的模块，使其方法可以相互协调并行地执行，或者让长周期的方法并行执行。

现在有很多的模式用来解决上述挑战。其中有些是专门应用于某个领域的，有一些甚至直接定义了每个组件的具体职责。如果要在这一章将所有的这些模式都介绍到，就超出了我们的分布式

计算模式语言的范围了，所以我们只是给大家推荐一下相关的文献，它们是[Fow97][PLoPD1][PLoPD2][PLoPD3][PLoPD4][PLoPD5][Ris01]。我们在第18章适配与扩展为大家展示了有关构建可适配和可扩展的组件的模式。当然，适应性和扩展性这个主题跟分布式系统的很多方面都有关系，而不仅是组件的实现。同样，支持并发执行的模式被我们放在了第15章中。

本章则仅关注于划分组件的模式，这些模式帮助我们解决与组件的分布相关的挑战。我们的模式语言中有6个这样的模式。

- **Encapsulated Implementation(封装实现)模式 (181)** 提供了一种基本的、典型的组件设计方式，使得组件既能够满足其接口契约职责，又不会通过接口泄漏其实现，因为通过配置或者创建选项所暴露的信息已经降到了最低。
- **Whole-Part (整体一部分) 模式 (183) [POSA1]** 让我们可以通过几个独立的、完备的内部对象组成组件对象。由此形成的组件对内部对象进行了封装，协调其合作关系，并为其功能提供一个公共的接口。客户端不能直接地访问内部的各个部分，对它们来说只能看到作为整体提供的功能。
- **Composite(组合)模式 (185) [GOF95]** 为那些由相似类型的对象组成的Whole-Part层级结构定义了一种划分组件对象的方式。这样客户端就可以统一地对待独立的对象和对象的组合体。
- **Master-Slave(主-从)模式 (186) [POSA1]** 支持容错、并行计算和计算精度。主组件将工作分配给一组从组件，并根据从组件的计算结果计算出最终结果。
- **Half-Object plus Protocol(半对象加协议)模式 (188) [Mes95]** 将在多个地址空间使用的逻辑对象组织成两个或多个协作的“半对象”。每个半对象实现这个组件功能的一部分。半对象之间通过同步协议进行协调。
- **Replicated Component Group(复制组件组)模式 (189) [Maf96]** 通过客户端透明的组件复制来达到容错的目的。复制的组件实现位于不同的网络节点，并组成一个组件组。客户端通过一个访问点与组件组进行交互，看上去就像只是一个组件一样。

你会看到不论是在本章还是在其他章节，甚至是其他地方，Encapsulated Implementation模式总是作为我们的模式语言中所有其他组件划分模式的通用的集成点(integration point)。特别是，Encapsulated Implementation既恰当地解决了如何实现组件的具体职责的问题，又没有引入任何性能、伸缩性、可用性和扩展性等操作上的新问题。因此，在我们开始实现一个具体的组件的时候它是最关键的一个模式。

图13-1展示了Encapsulated Implementation是如何与我们的分布式计算模式语言联系在一起，以及它是如何通过与其他模式的集成来达到高质量的组件划分和实现的。

本章接下来的两个模式，Whole-Part和Composite指导我们如何在组件内组织Whole-Part这种层级结构的对象。图13-2展示了它们在我们的分布式计算模式语言中的集成关系。

总之，Composite虽然非常具体，但是它确实是一种广泛应用的Whole-Part的形式。它们之间的主要的区别是Whole-Part可以用于组织完全不同的对象的层级结构，而Composite所管理的对象的职责、接口和属性往往是相似的甚至完全相同的。

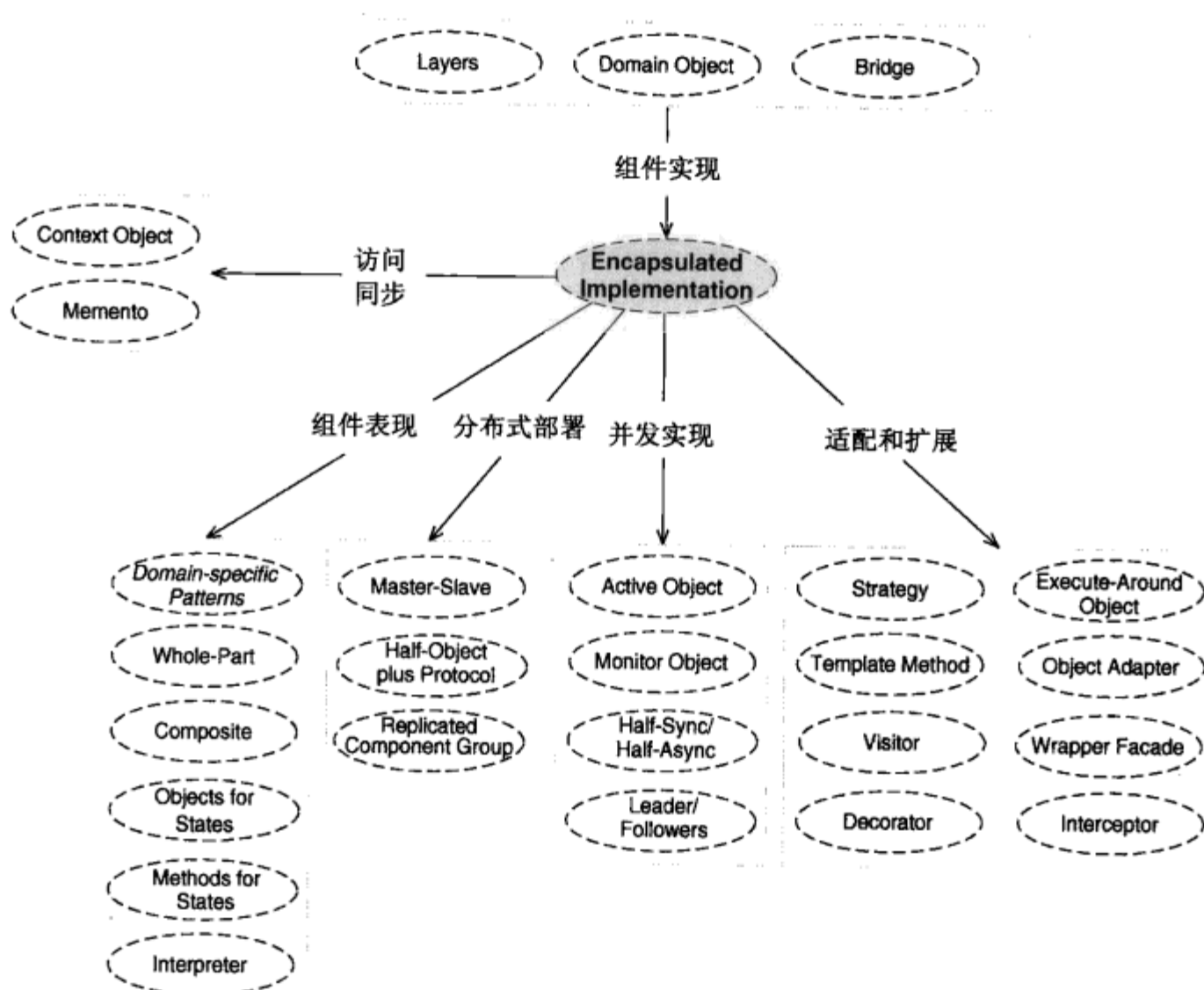


图 13-1



图 13-2

剩下的3个模式，Master-Slave、Half-Object plus Protocol和Replicated Component Group是关于组件的分布式部署的，目的是使得组件的性能和可用性等操作质量（operational qualities）得到提高。图13-3展示了它们是如何集成到我们的分布式计算模式语言中的。

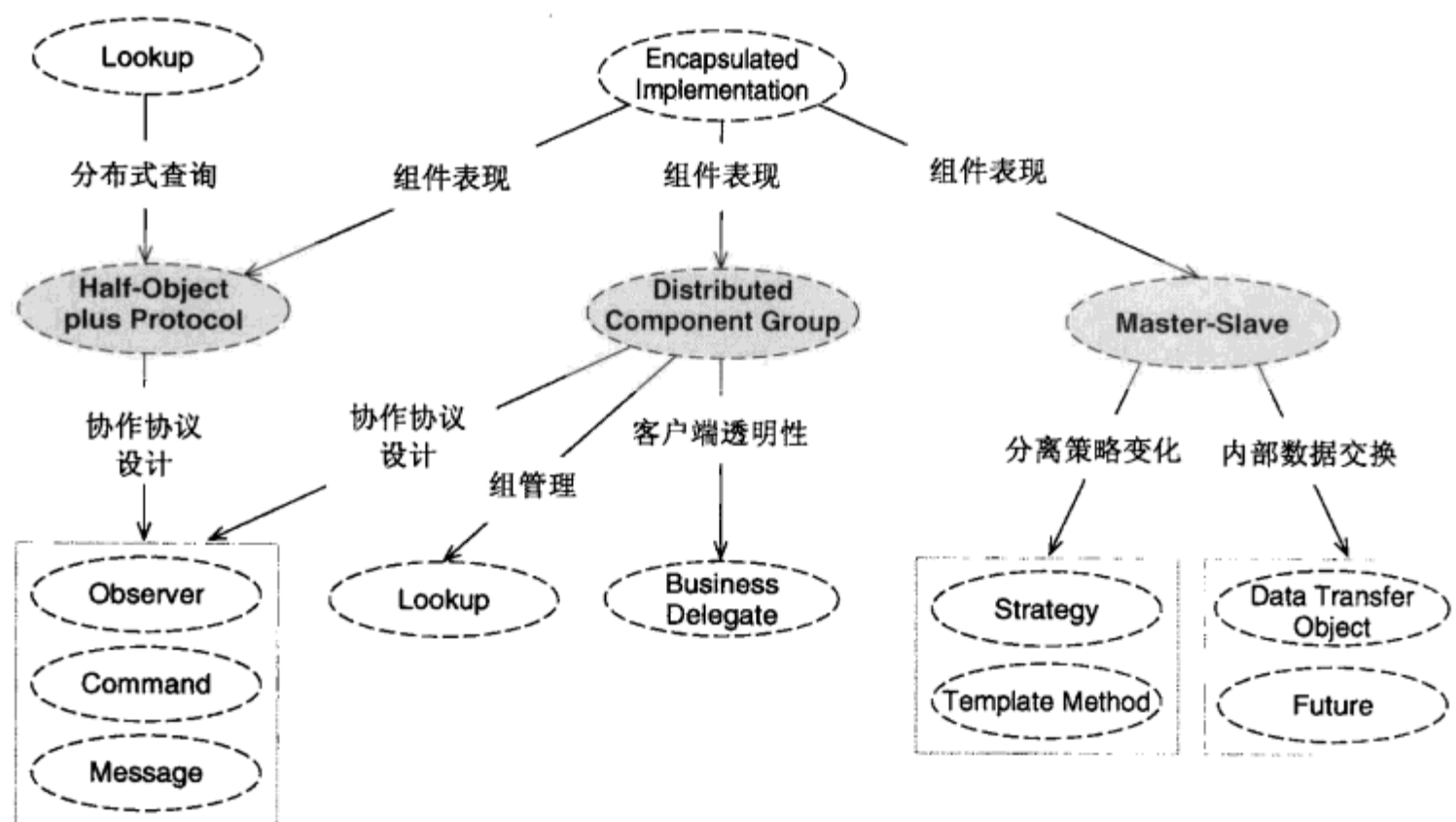


图 13-3

13.1 Encapsulated Implementation**

在开发Layers (108) 架构、应用的Domain Object (121) 和Bridge (255) 的时候，或者笼统地说，在为基于组件的系统设计组件的时候……如何根据组件的结构对组件进行实现是非常重要的——一件事。



组件通过自己的接口对外提供服务，接口定义了组件的使用协议、发布的功能和服务质量属性。然而，接口只是一个承诺，组件必须通过实现来履行这个承诺。因此，预设、约束等各方面的考虑虽然不能通过接口对外暴露，但正是这些因素制约着组件的实现。

除了由组件的接口定义的契约外，组件的实现还要考虑各种约束和需求，虽然组件的用户对这些并不那么感兴趣，但是它们对组件满足其契约却是至关重要的。例如，组件可能需要采用分布式的部署，而这种部署方式不能降低组件的性能和吞吐量。根据其具体应用和部署的情景的不同，系统可能要采用不同的算法，提供额外的功能或者更高的质量。比如，有时候我们需要使用不同的赋税计算方法，增加针对某个用户的业务活动，或者采用更强的安全措施。最后，几乎所有的组件都会经历升级：随着时间的推移，我们需要改进其内部的设计。尽管如此，这些都应当对客户端产生影响——它们关心的是契约，而不是如何满足这些契约。

因此，确保将所有组件的实现细节隐藏在其接口的后面，使得用户不会接触到你所选择的具体实现，因为这些选择可能会在应用的生命周期内发生变化，或者它们是依赖于组件的特定部署方式的（见图13-4）。

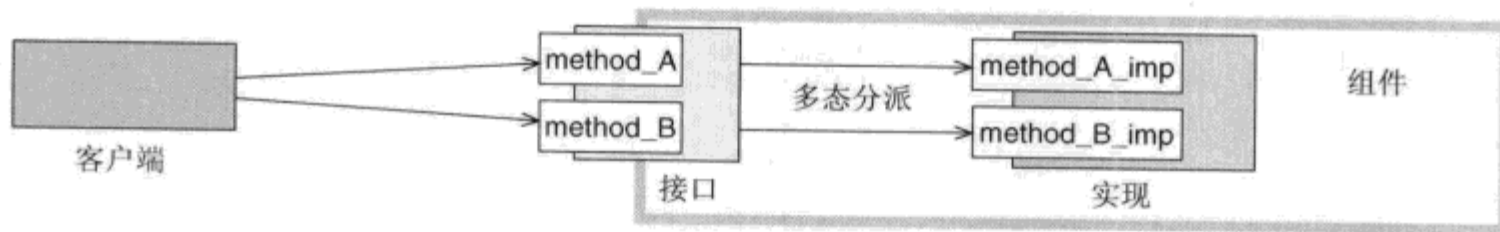


图 13-4

组件的客户端在编程的时候必须严格地依赖于正式公开的接口，这样才能保证对组件的实现所做的修改不会影响到客户端。



组件的实现也应该严格地遵守由接口定义的边界，这样才能保证其客户端可以依赖接口，完全依赖于全部接口而别无其他。Encapsulated Implementation可以自由地升级，而不会影响到客户端的稳定性。客户端代码的开发人员所使用的只有这个简单而稳定的接口。

接口边界封装了组件的实现使其不必关心其环境，反过来它所在的环境也不关心接口背后的实现。但是有时候组件却不可避免地要依赖于它的调用环境。为了避免引入反向依赖，在组件需要这种信息或者行为的时候，从其调用者向组件传入一个Context Object (243)。组件有时候也需要向客户端传回组件的某些依赖于实现的状态，比如在一次遍历中我们可能要传回当前的位置，或者为关心的已注册事件传回回调句柄。这种情况下可以通过返回Memento (242) 来保持组件的封装性。

总之，Encapsulated Implementation应该为组件的特定的功能职责提供一个封装良好的软件表示。有一些专属于某个领域的模式可以起到很好的帮助作用，比如健康管理、企业财务、电信和公共交通应用等，详情请参考[Fow97][Ris01][PLoPD1][PLoPD2][PLoPD3][PLoPD4][PLoPD5]。

有时候，组件功能的表示也可以通过一些通用的模式来定义，这些模式可能并没有绑定到某个特定的（应用）领域上。例如表现某些元素的层级结构的组件，根据其内部元素之间性质、行为区别的大小可以选择使用Whole-Part (183) 或者Composite (185) 设计。如果某个组件的功能集中于一个状态机，则根据其状态机的大小不同，以及各个状态之间共享的数据和上下文信息的多少可以选择Object for States (274) 或者Methods for States (275) 设计。如果组件的职责是解释某个结构化的文件或者某个给定语言中的句子——比如一个解析器，则可以考虑使用Interpreter (258) 设计作为其基础结构。

Encapsulated Implementation设计中一个关键的考虑因素是要满足其契约中指定的运作属性，比如性能、伸缩性等。仅提供“正确的”结构和行为并不能满足组件的可用性，自然也就无法通过验收了。能够帮助我们满足这些操作属性的两项关键技术是分布和并发。

将Encapsulated Implementation部署到分布式系统中的多个主机上面，我们就可以充分地利用整个网络中的资源，而不是局限在某个网络节点之上。可用的资源越多，组件的运作属性当然也

就越好。到底哪种属性能够从分布式部署中获益，要看你选择的是哪种组件划分模式了。Master-Slave (186)、Half-Object plus Protocol (188) 和 Replicated Component Group (189) 在性能、可用性、可伸缩性、容错和计算精确度方面做了不同程度的取舍。

并发的 Encapsulated Implementation 确实会对组件的操作属性产生很大的影响，尤其是其性能和吞吐量，因为我们可以同时处理多个客户请求。Active Object (212) 将组件的实现放到自己的线程里面运行，而 Monitor Object (214) 则将组件的实现放到客户端的线程里面运行。Half-Sync/Half-Async (209) 和 Leader/Followers (211) 适合于处理网络 I/O 的组件。

然而，上述部署和并发模型都是有代价的，主要包括功能重复、资源消耗增加、实现复杂，并且会引入 Encapsulated Implementation 的并发的各个部分或者与其他分布式组件之间的协调工作。所以在采用任何一个模型之前都应当仔细权衡，以免得不偿失。

除此之外，升级、扩展和适应能力也是几乎所有的 Encapsulated Implementation 所必须重点考虑的。因为这些开发属性对于组件能否在不同的部署环境中应用，甚至在完全不同的应用中重用都至关重要。

使用 Strategies (266) 和 Template Method (265) 可以将变化的行为同不变的行为分离开。Strategy 使用的是委托的方式，而 Template Method 使用的是继承的方式。相比之下，Visitor (261) 可以将最初开发的时候没有想到的功能添加到 Encapsulated Implementation 中去。组件内部的控制流可以用 Interceptor (260)、Decorator (262) 进行扩展，如果使用 C++ 实现 Encapsulated Implementation 的话，Execute-Around Objects (264) 也是个不错的选择。Interceptors 可以将“带外”(out-of-band) 的行为注入到函数的控制流内，而 Decorator 则用于包裹函数，在函数的执行前后完成其他的操作。Execute-Around Object 在这一点上和 Decorator 是相似的，它可以在一个语句序列的前后以线程安全的方式执行附加的功能，这使得它成为 C++ 用来获取和释放资源的惯用法。最后，Object Adapter (256) 和 Wrapper Facade (269) 支持将 Encapsulated Implementation 集成进它所在的环境里面，它们可以将组件、库和操作系统提供的接口适配成 Encapsulated Implementation 期望的或者要求的接口。这两种设计的区别在于 Object Adapter 不会隐藏被适配的接口，这些接口仍然可以被 Encapsulated Implementation 访问，而在 Wrapper Facade 的设计中这些接口完全被封装起来了。

在具体的应用中使用 Encapsulated Implementation 的时候当然也需要一定程度的灵活性，然而过度的灵活性可能会适得其反，组件可能会灵活到完全没有用的地步 [Bus03]。因此，在 Encapsulated Implementation 中只能支持强制的变化，而非所有的“最好是有”的变化。要辨别哪些变化对于组件来说是强制的变化，可以参考我们列出的这些资料中的方法：*Open Implementation Analysis and Design* [KLLM95]、*Commonality/Variability Analysis* [Cope98] 和 *Feature Modeling* [CzEi02]。

13.2 Whole-Part**

在对 Encapsulated Implementation (181) 进行划分的时候……我经常需要或者说可以将复杂的组件对象分解成几个小一点的块。



有些组件对象提供的功能非常之多，这时候将这些功能捆绑在一起往往是不切实际的。我们最好把它们分割开，让每一部实现一部分职责。尽管从开发的角度看确有划分之必要，客户端却并不想与这么多小的块打交道。

尤其是组件不应当允许客户端直接访问其内部的各个部分，因为基本上它们不能单独为客户端提供有意义的服务。而且，如果对这些小块进行重新划分，客户端不应当受到影响。另一方面，我们设计这些小的组成部分时，需要考虑到让其可以在其他地方重用，这就要求必须避免对某些组件的紧耦合，因为每个组件可能以不同的方式来集成这些部分，或者跟其他的部分集成。

因此，将组件对象看作一个Whole（整体），将其划分为多个独立的Part（部分），使用Whole对其进行封装。为Whole定义一个接口，这个接口就是客户端访问组件的唯一入口（见图13-5）。

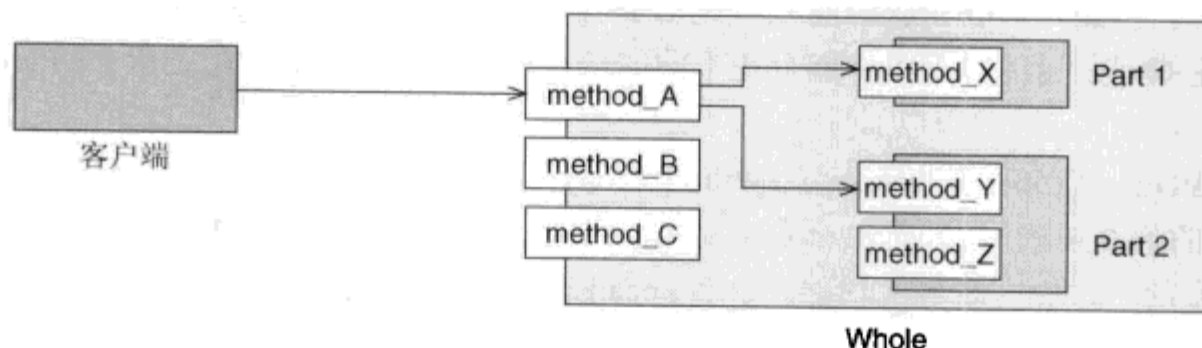


图 13-5

这些Part本身实现了自我完备的实体或者机制，它们可以用来构成更高层的功能。相比之下，Whole代表的是一个集合它实现了一个policy，其功能由前面所说的Part提供。当客户端调用Whole上的方法时，方法会执行其policy，这个policy使用由Whole封装的一个或多个Part来实现其行为。



Whole-Part设计可以防止客户端直接访问集合（aggregate）中的零件，它使得Whole对外表现为一个自我完备的语义单位。在Whole-Part设计中，我们把policy和零件（mechanism）进行严格的划分，这样集合组合使用各个组成部分的时候，不论是对于客户端还是Part本身都是透明的，同时我们也就可以在其他的集合组件中重用这些Part。

实现Whole-Part结构有很多方式。Whole经常使用Mediator (239) 或者Facade (171) 实现，而Part则往往本身就是独立的组件。为了避免遍历多个Part的功能分散到各个Part里面，我们可以使用Visitor (261) 模式来实现按照某个顺序完成遍历。每个Part本身又可以是一个Whole-Part结构，这样就可以构成一种递归的Whole-Part设计。在Whole和Part之间的数据交换往往可以用Data Transfer Object (244) 封装，以避免Whole依赖于具体的数据表现，方便Part在其他的Whole-Part结构中的重用。

在实现Whole-Part结构时需要考虑的两个重要的问题是对Part的共享及其生命周期管理。在很多设计中，比如工厂流程控制系统的传动装置的表现，Part不存在与其他Whole共享的问题，而且从设计的外部来看也是不可见的。因此，Part的生命周期是绑定在它们所构成的Whole的生命周期之上的——它们在Whole创建时创建，在Whole销毁的时候销毁。而在有的Whole-Part结构

中, Part的生命周期是独立于Whole的, Part可以在多个Whole中共享——比如Email的附件。这样的Part必须能够自我管理生命周期, 包括它们绑定到一个或多个Whole上时的所有的约束。在不支持Automated Garbage Collection (307) 的环境中, 我们可以用Counting Handle (309) 来辅助管理独立的和共享的Part的生命期。

13.3 Composite**

在实现Encapsulated Implementation (181) 或者Interpreter (258) 配置的时候……有时候我们需要按照统一的方式对待原子元素和Whole-Part结构中的组合元素。



有时候整个层次结构由支持相同的接口的对象以递归的方式构成。然而, 客户端通常并不关心具体的组织形式及其结构中的递归特性, 它们只希望将整个层次结构作为一个整体来使用和交互。

尽管如此, 在这个整体的内部我们必须保持这种层级结构。比如, 它可能代表应用领域内某种层次关系, 例如一个网络或者仓库的拓扑结构, 或者文件系统的分层结构。更为麻烦的是, 这种层级结构内部的对象重新组合, 比如, 将文件系统结构中的一个文件或者文件夹移动到另一个文件夹或者其他的卷上的时候, 其对于客户端的透明性不应当遭到破坏。而且, 如果要扩展这个层级结构, 增加一个新的实现了同一个共用接口的对象类型, 不应当对已有的其他类型产生太大的影响。比如, 我们可能需要向文件系统结构中增加一个回收站或者新的文件类型, 此时文件系统的层级结构必须是能够保持稳定的。

因此, 声明一个组件接口, 通过这个接口来确定层级结构的整体中所有对象的共有职责, 并通过对这个接口进行子类化来实现整体内部的具体对象, 以及由这些对象递归组合而成的对象(见图13-6)。

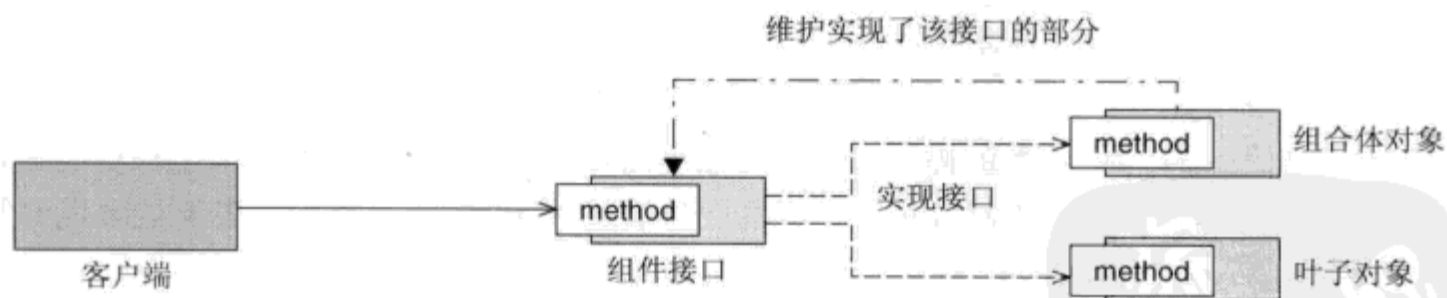


图 13-6

Composite层级结构中的对象可以分为两类: 叶子和组合体。叶子对象实现了原子实体的行为, 并且不可再分, 具体的文件类型就是一个叶子的例子。组合体对象定义了层级结构中聚合在一起的多个对象的行为, 以及维护多个支持共享的接口的对象的职责, 比如文件系统中的卷和文件夹。

客户端通过组件的接口来引用和管理该聚合体。如果该接口代表的是一个叶子, 请求就会通过叶子的实现执行。相反, 如果接口代表的是一个组合, 请求会被传递给它的一个或多个孩子,

如果孩子是叶子就会直接执行这个请求，否则继续递归地将请求传递给自己的孩子。而且，在将请求传递给孩子之前或者之后组合都可以执行自己的活动。对组件接口调用的结果沿着这个由叶子和组合构成的递归调用链依次返回。



Composite设计可以用于表现任意的Whole-Part结构，只要其内部的对象实现了组件的接口。而且，该结构对于客户端是透明的：客户端只能看到所有的对象遵守一致的契约。而且，这个接口也易于升级，对已有的Composite配置进行扩展或者重组不必对已有的叶子和组合进行修改。另一方面，Composite设计也只有在Whole-Part层级结构中所有的对象实现相同的职责时才有用。为了避免客户端看到Composite的层级结构特征，其叶子和组合的所有的方法都要集成到组件的接口中。所以，不同的方法越多，组件的接口就越臃肿，就会有很多功能仅由有限的几个叶子和组合对象实现，这样该接口对客户端来说意义也就越小。

通常我们都是使用Iterator (173) 或者Enumeration Method来访问Composite内部的元素，这样可以避免依赖于Composite结构的内部具体实现形式。Visitor (261) 可以用于实现对整个Composite的层级结构的访问，从而避免将要实现的功能同遍历对象层级结构的策略紧密地耦合在一起。

13.4 Master-Slave**

在实现Encapsulated Implementation (181) 的时候……我们经常需要某种措施来提高系统组件实现的性能、容错能力或者结果精度。



有些组件必须满足很高的性能、容错能力或者计算精度方面的要求，尤其是那些处理关键和复杂应用的组件，比如核动力工厂控制系统，或者是医学系统中负责DNA分析的服务。使用更多、更强大的计算资源当然会有所帮助，但是类似的解决方案往往过于昂贵，而且有时候也不充分。

例如，通过优化算法和使用更快的硬件可以提高性能，但是对于DNA分析这样的要处理大规模数据的NP困难或者NP完全问题，就未必能帮上多大忙了。我们可以通过对算法和代码的检查提供计算精度，但是对于复杂的算法来说这种所谓的检查几乎是不可能的，或者至少存在出错的可能性。代码的稳定性和健壮性的提高可以提高系统的容错能力，而对于组件宿主环境的故障恐怕就无能为力。然而组件必须能够以可以接受的代价满足上面的需求、解决这些问题才能够算得上是有用的。

因此，采用“分而治之”的策略，来满足性能、容错或者计算精度方面的需求。将服务划分成独立的、可并行执行的子任务，通过合并这些子任务的返回结果来得到服务的最终结果（见图13-7）。

我们对组件进行适当地划分，使其包括一个Master和至少两个Slave，Master实现了某种“分而治之”的策略，并作为客户端访问组件的入口；Slave实现了可以并行执行的子任务。Master收到来自客户端的请求，它将工作按照可用的Slave的数量分成若干个部分，将每一部分的处理委托给一个独立的Slave，等待所有的Slave执行完毕，从它们返回的部分结果中计算出最终结果并返回给客户端。



图 13-7

由于各个子任务可以并行地执行，Master-Slave设计对于提高长周期服务的性能尤为明显。因此，Master-Slave在多核处理器上部署的组件中应用非常广泛而有效。Master-Slave配置支持的其他操作属性还包括容错、可靠性和计算精度。我们可以通过运行Slave的多个副本来支持容错和可靠性，这样即使一个Slave出现故障，其他的Slave仍然可以继续执行请求并返回结果。计算精度可以通过多个实现了同样服务的Slave——通常采用不同的算法——同时运行，并对它们返回的结果以表决的方式实现。

Master-Slave配置的缺点在于，只有当相关的服务可以采用“分而治之”的策略拆分为几个并行的子任务的时候，我们才可以应用。例如，如果要处理的数据具有高度的独立性，我们就可以将其分为多个同样大小的块，然后用多个具有相同功能的Slave对其进行处理。对大型的或集群数据的操作属于此类情况[DeGe04]。如果数据不能进行划分，也可以尝试将服务本身划分为若干个独立的步骤，并通过一系列Slave并行地执行。很多图形方面的算法属于此类，尤其是那些用于路由和收集状态信息的算法。如果对服务所进行的划分需要在子任务之间进行协调和同步，使用Master-Slave来满足组件的性能、容错能力或者精确度方面的需求可能就不大合适了。

在Master-Slave结构中，Master是客户端对组件集中的访问点，封装了组件的“分而治之”的策略。这个策略主要是由Master-Slave结构的意图决定的。如果我们关注的是性能，Master就应该将要处理的任务或者数据根据可用的Slave的数量划分成尽量多个相似的块，并让所有的Slave并行执行，然后根据每个Slave返回的部分结果计算出最终结果。如果组件的意图是提高计算精度或者容错能力，就可以让所有的Slave处理整个数据。然后采用类似“m个Slave中的n个必须返回同样的结果”这样的策略来帮助Master决定最终结果。我们可以使用Template Method (265) 或者Strategy (266) 来实现在不影响组件客户端、Slave，并且在Master对Slave进行协调的核心算法的情况下，为组件配置不同的“分而治之”的策略。

大多数的Master-Slave结构中都只有一个Master。然而这种设计会引入单点失败的问题，这有时候是不可容忍的。所以我们还有一种实现方式是将Master和Slave的角色合并到一个对象上面，然后通过指定哪个具体的实例代表什么样的角色来定义Master-Slave结构。使用心跳等合适的机制，Master可以周期性地通知所有的Slave它还活着。如果某个Slave没有收到这样的通知，它就假定Master死掉了，随后它会声明自己扮演Master的角色，并将这个接任的消息通知剩余的Slave。通过与底层通信基础设施的配合，随后的客户端请求就会发送给新的Master。当然，我们也可以

用看门狗来监视Master的状态，在其死掉之后重新启动Master。

在Master-Slave结构中，Slave的个数很大程度上取决于可用的内存、线程和CPU等计算资源的数量。可用的CPU越多、并发能力越强，在Master-Slave结构中指定的Slave也就越多。

在Master和Slave之间的数据交换通常使用Data Transfer Object (244) 来进行封装，目的是避免依赖于特定的数据表现。对Slave的计算结果的访问可以使用Future (223) 来方便地进行协调。在调用Slave的时候，它会返回一个Future来代表将计算出的结果。在Slave完成计算之后，它便使用相应的数据来填充Future。如果Master在相应的Slave填充该Future之前对其进行访问，它将被阻塞在那里直到结果有效。

13.5 Half-Object plus Protocol**

在实现Encapsulated Implementation (181) 或者Lookup (292) 的时候……我们经常需要减少从多个地址空间访问某一个对象的响应时间。



在分布式系统中，我们的设计经常需要允许客户端访问位于其他地址空间的组件内的对象。然而，由于在跨网络交换请求和响应的时候经常会引入延迟和抖动，这使得我们的设计不能适应快速响应的要求。

要想通过复制的方式来解决这个问题，并不总是那么有效。比如，组件可能需要访问来自多个地址空间的信息来执行自己的动作。如果要从每个复制的组件获得这些信息，会引入很大的网络负载和流量，这就会减少我们从组件复制获得的性能提升，甚至可能会降低系统的性能。如果复制的组件代表的是分布式系统中的一个单状态实体，也会遇到类似的问题。如果这些复制组件中的任意一个的状态发生了变化，所有的其他的复制组件必须做相应的更新，而且这种更新往往是跨网络的。

因此，将这些对象分成多个“半对象”，每个半对象在哪儿使用就放在哪个地址空间中。每个半对象只实现它所在地址空间内的客户端所要求的功能和数据。在所有的半对象之间通过一个协议来帮助它们协调其动作，并保持其状态的一致性（见图13-8）。

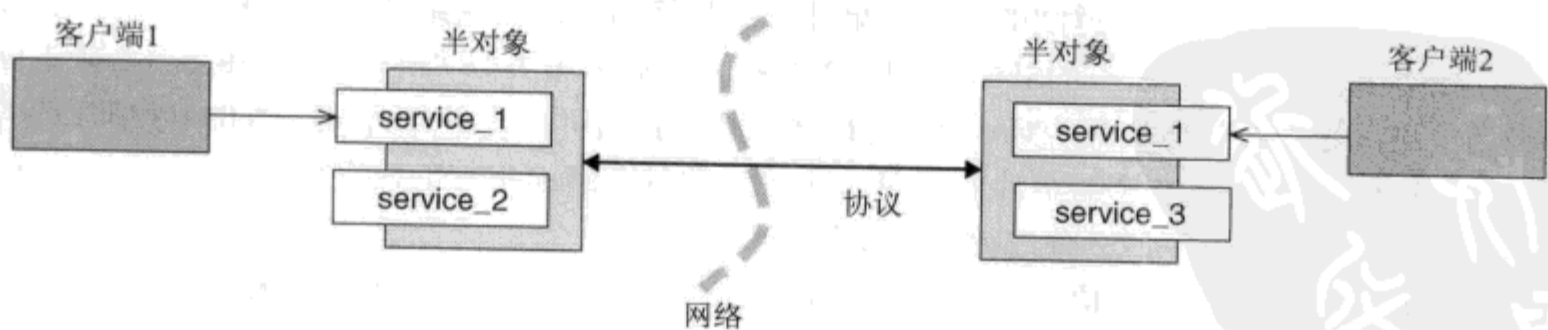


图 13-8

如果客户端要调用组件上的某个服务时，发现它不需要其他地址空间内维护的信息，相应的半对象就可以直接在客户端地址空间的本地执行，而完全无需通过网络。否则，在客户端本地的半对象通过连接分布的半对象的协议从系统的其他地址空间获得必要的信息。

◆◆◆

Half-Object plus Protocol结构通过降低对网络的应用优化了系统的性能，但是其代价是功能和/或数据的重复。所有可以在客户端地址空间本地执行的逻辑对象的服务由相应的半对象在本地执行。仅当在服务执行时需要多个地址空间的情况下，才通过协议将必要的半对象连接起来共同完成分布式计算。总的来说，在客户端地址空间可以本地执行的功能越多，需要在多个半对象之间交换以保持其一致性的数据越少，Half-Object plus Protocol的性能就会越好。需要分布式计算的内容越多，需要通过协议交换的数据越多，Half-Object plus Protocol就变得越鸡肋。总的指导原则就是，尽量地减少内部状态的复制，从而减少通过协议进行数据交换和同步的必要。

Half-Object plus Protocol还带来一个附加的好处就是——可伸缩性。每加入一个新的半对象，都同时加入新的地址空间，这就避免了使用其他的地址空间内的计算资源。

半对象间的协议的具体设计取决于需要什么样的协调机制。简单的数据交换协议可以基于Observer (237) 实现，它可以避免不必要的更新和协调动作。该结构中的半对象互相调用的动作可以封装成Command (240) 或者Message (245)，以确保协议独立于特定的动作类型。

13.6 Replicated Component Group**

在实现Encapsulated Implementation (181) 的时候……我们有时候必须为组件的实现提供容错能力和高度可靠性。

◆◆◆

系统中的有些组件必须满足高可靠性和高容错性的需求，尤其是那些复杂执行和协调集中活动的组件，比如在电信系统中的目录服务。像热备份或者冷备份之类的蛮干式的解决方案，因为拥有它需要过高的总计成本，所以对于很多应用来说都过于奢侈了。

在大多数系统中，只有少数几个组件需要极高的可靠性和容错能力，使用基于硬件的系统级解决方案有些不值得。然而，要在一个低成本的环境中提供高可靠性和容错能力确实是个挑战。标准的硬件和网络本身就是故障之源，它们都可能导致系统处于不可用的状态。因此，要求高可靠性和容错能力的组件必须明确地考虑到：网络连接和进程间交互可能会失败，主机可能会宕机。简单地说，组件必须在一个虚弱的环境中保持自身的强壮。

因此，提供一组——而不是一个——组件的实现，将这些组件实现的复制件分布到不同的网络节点上。将客户端在组件接口上的请求转发给所有实现的实例，并等待其中的某个实例返回执行结果（见图13-9）。

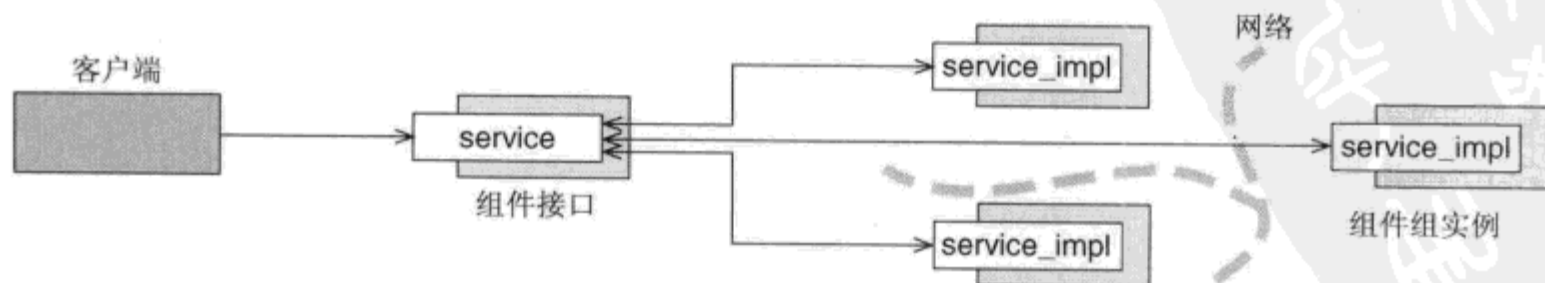


图 13-9

在组件组所有的实例中返回的第一个结果即返回给客户端。



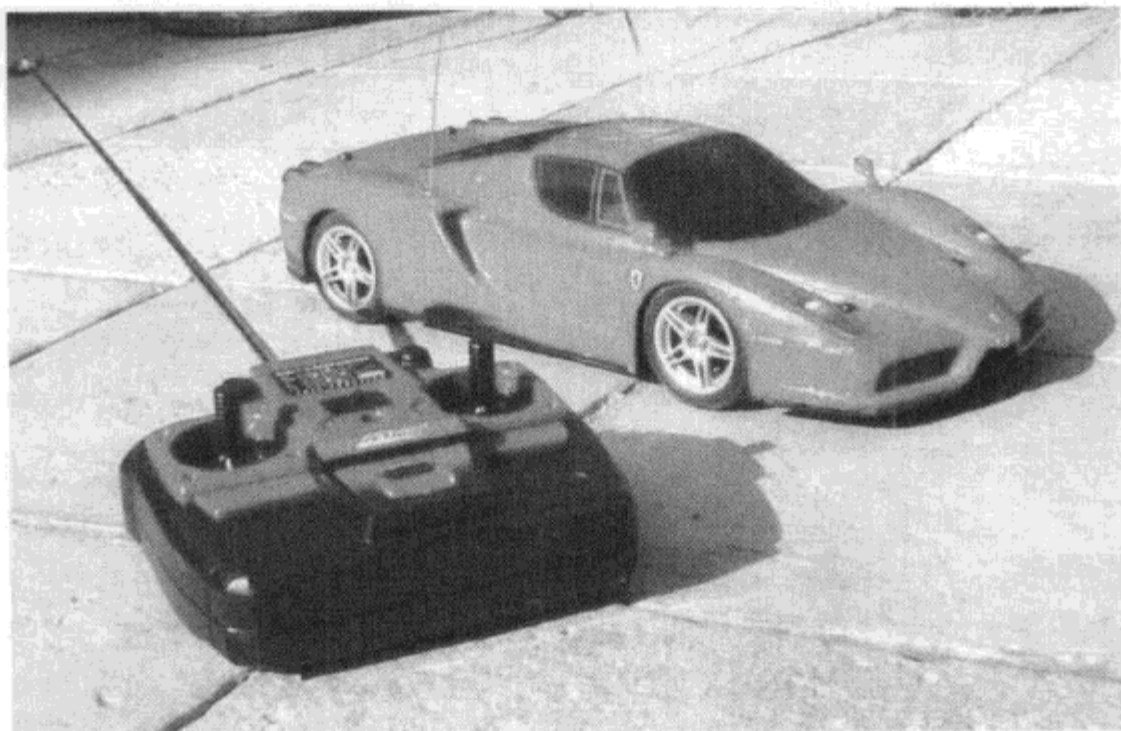
Replicated Component Group最大的好处是它增强了组件的容错能力，只要组件的实例中至少有一个可以访问的，客户端的请求就可以得到执行。Replicated Component Group也提高了系统的可靠性，因为如果组件组中有一个组件的实例因负载过重不能及时处理请求，其他的低负载的对象的实现仍然可以服务这个请求。

Replicated Component Group结构的一个缺点是它要求所有组件的实例要维护一致的状态，这可能会导致“絮叨”的通信，并引入大量的网络流量。要维护的状态越多，变化的比例越高，要保持组内状态一致所必需的网络负载就越多。而且，对Replicated Component Group的一个请求会由组内所有的组件实现同时执行，这样就会消耗大量的计算资源。所以，Replicated Component Group设计只适用于分布式系统中扮演关键角色的组件，这些组件的可靠性和容错能力对于系统的可操作性而言应该是至关重要的。

我们可以使用Business Delegate (170) 作为组件的接口，帮助保持Replicated Component Group对其客户端是透明的。额外的Lookup (292) 功能使得组件的实现可以动态地加入和从组中剥离，这样我们就可以在运行时重新部署组件组的成员，而不会影响到组件的可靠性。

保持组内实例状态一致性的协议往往是基于Observer (237)，因为它可以避免不必要的更新和协调动作。这个设计中所执行的请求可以使用Command (240) 或者Message (245) 进行封装，这样可以保证与组件组的通信独立于特定的请求类型。





遥控小汽车
© Kevlin Henney

为了在网络上部署应用，我们需要把用户界面从应用的核心功能中分离出来。这种分离措施确保我们可以独立地修改和访问用户界面和应用的核心功能。然而在实践中我们怎样才能有效地分离这两个方面呢？本章提供的8个模式就是有关这个主题的。它们可以用于将用户输入转换成具体的服务请求，将服务请求的结果转换成用户输出，以及对应用功能的安全访问。

将用户输入转换成对应用功能的服务请求，执行服务请求并将结果转换成有意义的输出展示给用户并不是件容易的事情。如果用户界面和功能还是分离的，要做到这些就更麻烦了。之所以要分离用户界面和功能，目的往往是为了可以独立地升级用户界面和应用功能，简化底层软件和硬件技术的修改，使得应用可以分布式部署在异质的平台上。

要把应用的用户界面同其他方面的功能分离开需要考虑以下问题。

- 数据结构解耦。分离用户界面和应用功能意味着应用组件所使用的数据结构应当独立于负责控制和表现的数据结构。比如，在用户界面元素上的一次鼠标点击，应当转换成对

一个或多个应用组件的服务请求。同样，作为对服务请求的响应，返回的数据必须转换成某种适当的格式，以便能够在指定的输出设备上展现出来。如果不做解耦，想要不修改相关的应用功能而单独改变用户界面的外观和感觉几乎是不可能的。

- **位置解耦。**最简单的GUI框架通常是假设用户界面、应用逻辑和数据都放在一台机器上，而且只有一个访问用户。在分布式应用中，这种假设是不可想象的，因为分布式应用——特别是多层的系统——其用户界面往往是远程执行，应用逻辑和状态由多个用户共享，数据也往往不跟应用逻辑和用户界面放在同一个主机上。
- **工作流解耦。**很多应用是工作流驱动的，用户看到的是一个表单序列，或者在某个条件下能看到某些特定的表单。直接在用户界面或者应用逻辑中实现工作流逻辑会把两者耦合在一起。这种耦合的结果就是，修改工作流就得同时修改用户界面元素，修改用户界面元素又会影响到核心的应用功能。
- **技术解耦。**用户界面往往是使用特定的用户界面技术实现的，这些技术可能会独立于用户界面的外观和感觉而发生变化。如果我们预计到在应用的生命周期中用户界面技术会发生变化，做架构的时候就要考虑将用户界面中独立于技术的方面和依赖于某些技术的方面分离开。
- **对请求进行显式的协调和控制。**处理和执行对应用和组件的请求往往需要一些协调和辅助性任务。比如，我们可能需要调整来自多个客户端的调用的顺序；或者，需要支持日志功能和撤销/重做功能；或者，需要基于策略的失败处理方法。对于这些活动，它们的策略和配置应当依赖于特定应用实例的需要，而不应当作为应用的用户界面或者核心功能内置的属性。所以，我们应当保证用户界面和对组件的调用之间是相互独立的。
- **安全。**虽然能否安全地访问应用服务在用户验收中越来越受到重视，网络应用——尤其是可以在互联网上访问到的网络应用——面对攻击仍然显得非常脆弱。比如，恶意用户可能会试图访问和执行一些未被授权的功能。得到某个应用或者部分功能授权的用户可能有意或无意地在服务请求中嵌入攻击。鉴于用户界面往往运行在应用的客户端，用户是机器的主人或者用户拥有管理权限，这些机器往往容易被人利用，所以大多数网络应用的信任边界都出现在用户界面和核心（领域）功能之间。

上述问题必须得到妥善的解决。如果这些问题解决不好，修改用户界面时就会出现涟漪效应，影响到功能的实现，反之亦然。而且，这些问题之间经常是相互依赖的，为了解决某一个问题的解决方案可能会约束其他问题的解决方案。

我们的模式语言中的8个模式——大部分来自*Patterns of Enterprise Application Architecture* [Fow03a]——正是用来应对以上问题的。这些模式为我们提供了常见的引入上述分离控制的方式，有隐式的也有显式的。

- **Page Controller（页面控制器）模式 (196)** [Fow03a]为基于表单的用户界面中每个表单引入一个清晰的入口点——页面控制器，它将每个表单发出的服务请求的处理与执行联合起来。
- **Front Controller（前端控制器）模式 (197)** [Fow03a]为应用建立一个唯一的入口——前置

控制器，它将由用户界面发出的服务请求的处理和执行联合起来。

- **Application Controller**（应用控制器）模式 (198) [Fow03a]将用户界面的导航和应用的工作流控制分离开。**Application Controller**从应用的用户界面收到服务请求，根据当前的工作流状态确定调用哪项服务功能，然后根据服务的执行情况将相应的视图展示在用户界面上。
- **Command Processor**（命令处理程序）模式 (199) [POSA1]将服务请求与服务的执行分离开。**Command Processor**组件将请求作为独立的对象进行管理，调度请求的执行，为其提供日志、存储、撤销/重做等附加功能。
- **Template View**（模板视图）模式 (200) [Fow03a]为每个视图引入一个**Template View**组件，它可以使用某种特定的用户界面技术将应用数据或者其他信息用一种预先设定的视图格式展现出来。
- **Transform View**（转换视图）模式 (201) [Fow03a]引入了一个**Transform View**组件，它可以将为响应特定的用户请求而从应用接收到的数据转换成数据上的具体视图。
- **Firewall Proxy**（防火墙代理）模式 (202) [Sfhbs06]通过引入一个代理来阻止来自外部的攻击，这个代理可以检查服务请求，发现其中的可疑内容并将其移除。
- **Authorization**（授权）模式 (204) [SFHBS06]会检查客户端的访问权限，确保只有符合指定访问规则的授权客户端才可以访问应用的特定功能。

这些模式的功能正是为了实现将用户界面和核心功能分离开。关于核心功能的内容，我们在前面有详细的论述，尤其是**Model-View-Controller** (109) 和**Presentation-Abstraction-Control** (111) 两节。下面你会看到，我们前面列出的这些模式并非相互独立的，它们可以组合在一起互为补充，每一种组合用于处理某种常见问题的不同方面。

Page Controller和**Front Controller**两个模式的目的是减少访问应用功能的控制器数量。它们基本的区别是其范围不同。**Page Controller**适合于基于表单或者页面的用户界面，而且访问应用功能的工作流相对简单的情况，比如静态的HTML页面。**Page Controller**并未给每个可以通过表单或者页面调用的动作设计一个单独的控制器，而是使用一致的、统一的方式处理这些动作。相反，如果用户界面发出的服务请求在执行之前必须先转换成对应用功能接口的调用，则使用**Front Controller**效率会更高。通过HTTP协议发送给Web服务器的请求就是一个这方面的例子。

图14-1展示了我们的分布式计算模式语言中的**Page Controller**和**Front Controller**是如何与其他模式联系在一起的。

Application Controller和**Command Processor**主要是协调对应用功能的访问。**Application Controller**处理的是基于工作流的系统，系统根据当前的计算状态决定为某个服务请求执行哪项功能。**Command Processor**则支持根据优先级、截止时间等调度规则决定由多个客户端发出的请求的执行顺序。**Command Processor**的目的是满足某个特定方面的服务质量，或者使应用的吞吐量最大化。

图14-2展示了**Application Controller**和**Front Controller**是如何集成到我们的分布式计算模式语言中去的。

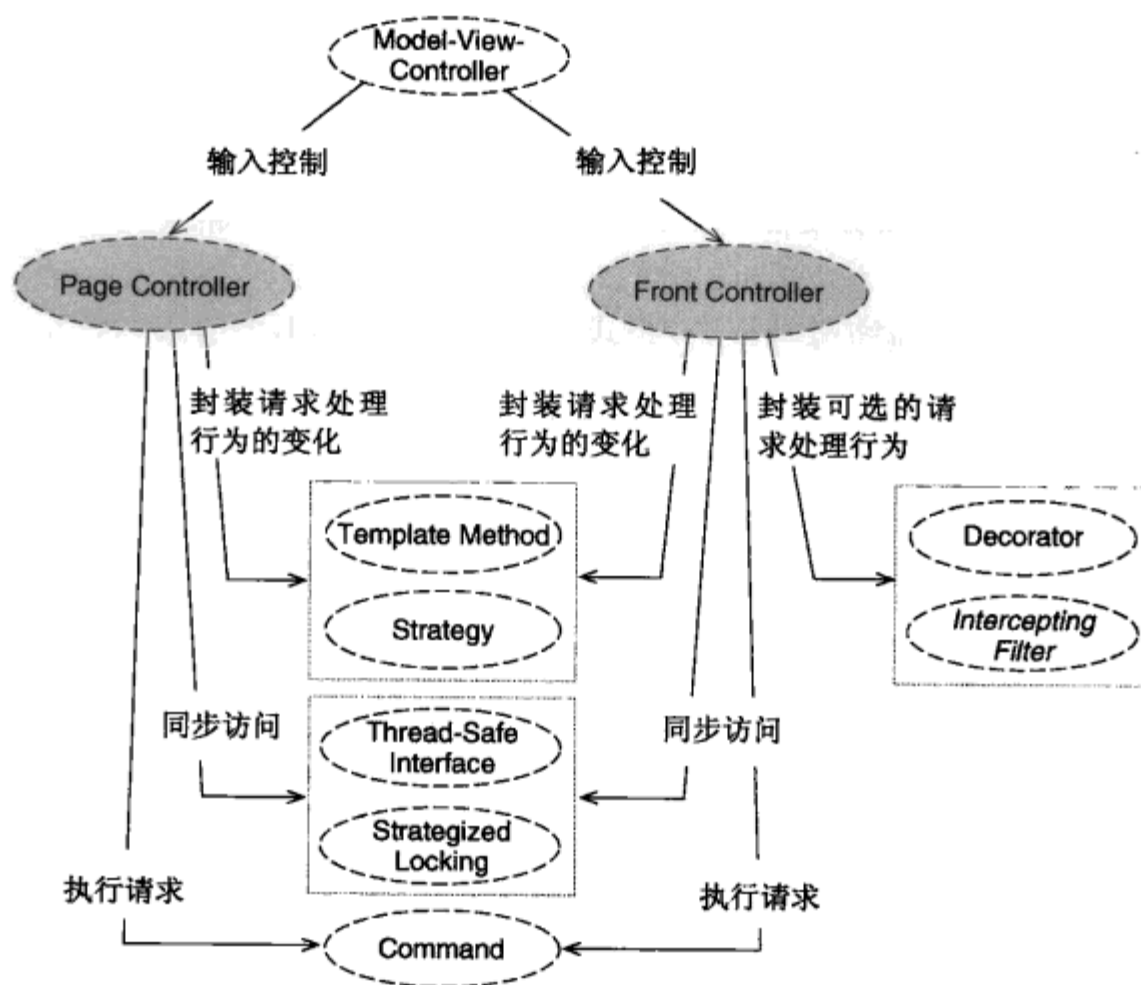


图 14-1

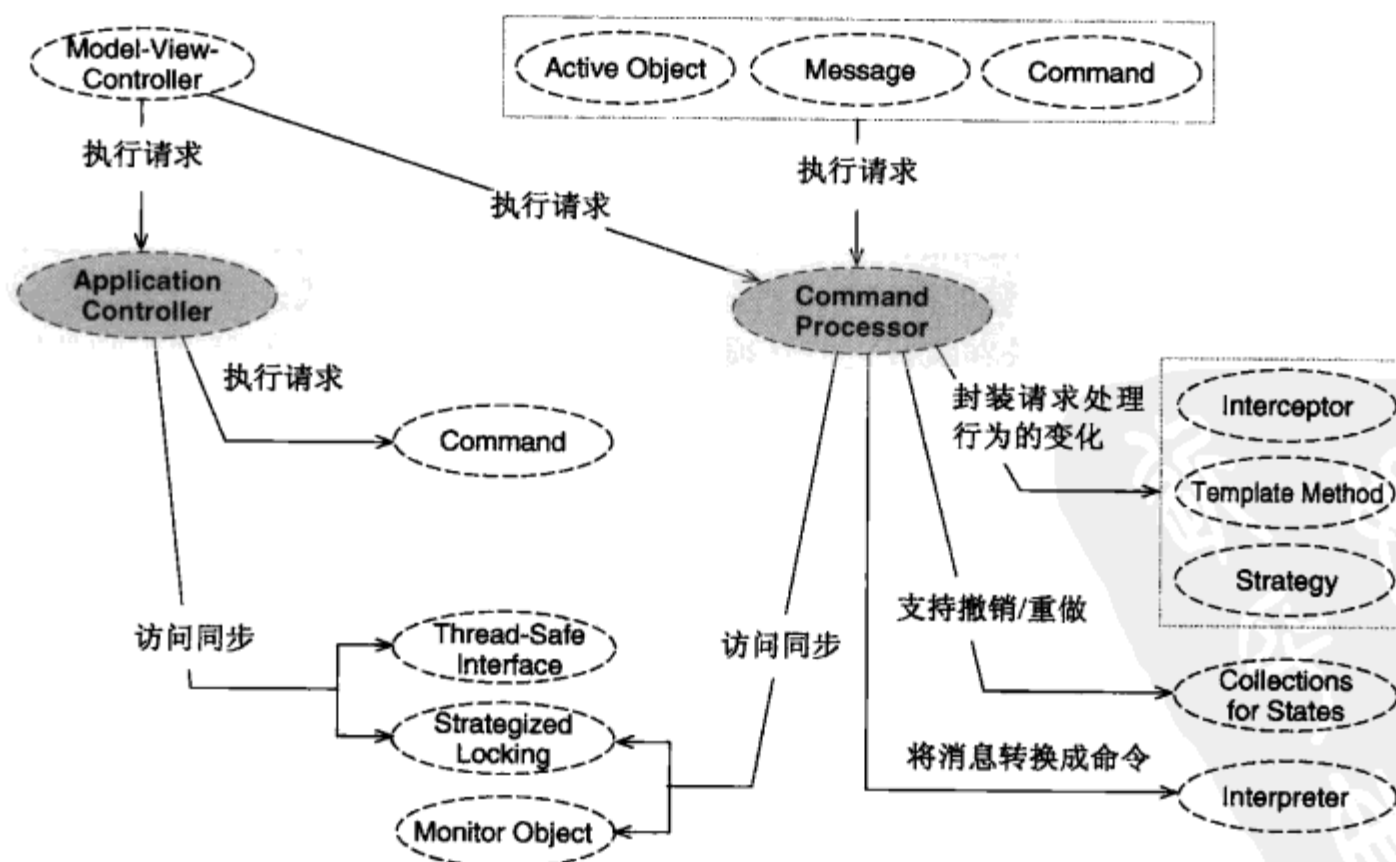


图 14-2

Template View和Transform View的目的是将应用数据转换成应用上面特定的视图。它们主要的区别是其解决问题的角度不同。Template View采用的是以用户界面为中心的角度：由JSP或者ASP.NET等具体的用户界面技术来决定如何从应用中获取数据。与之相反，Transform View则是采用以应用为中心的角度：首先从应用中获取数据，然后使用某种用户界面技术将其转换成特定的视图。

图14-3展示了Template View和Transform View是如何集成到我们的分布式计算模式语言中去的。

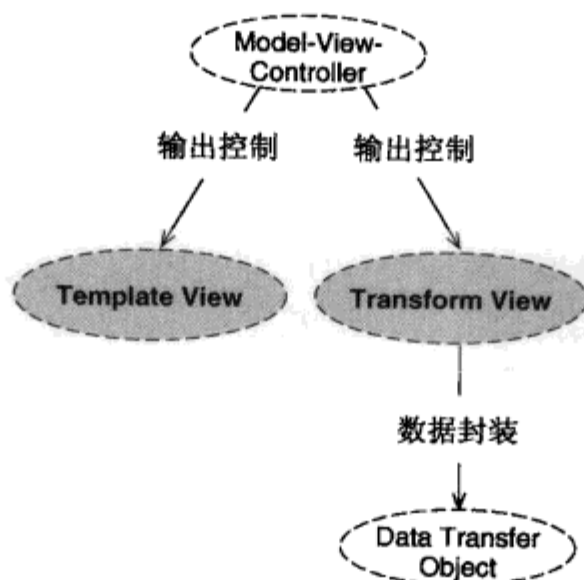


图 14-3

本章最后的两个模式，Firewall Proxy和Authorization用于处理对应用的安全访问中互为补充的两个方面。Authorization确保只有可信的客户端可以访问应用；Firewall Proxy确保这些客户端访问应用的方式是被允许的。图14-4显示了它们与我们的模式语言中其他模式集成在一起的情形。

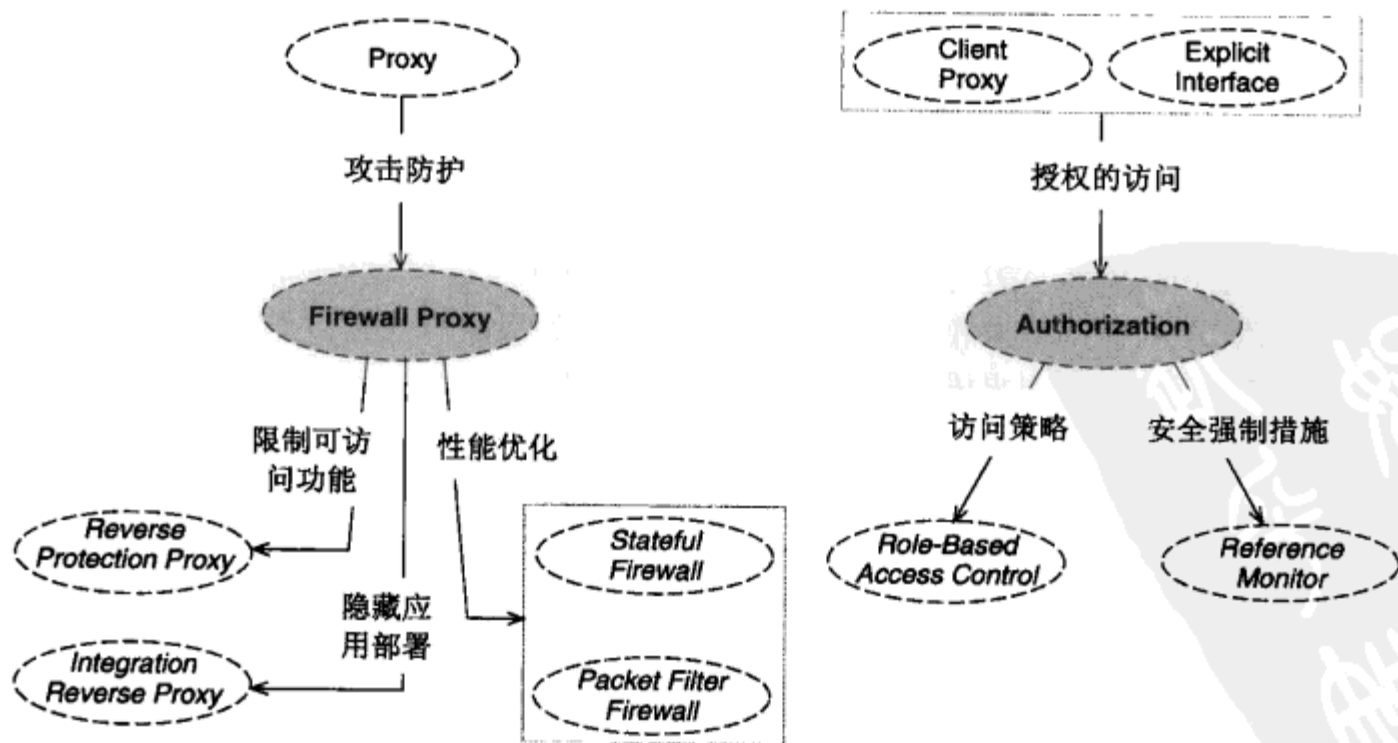


图 14-4

14.1 Page Controller**

在开发Model-View-Controller (109) 架构的时候, 如果视图相对于模型是远程的……则对于那些基于表单的用户界面, 我们需要一种机制能够将特定表单所发出的服务请求的处理和执行联合起来。



有些应用提供了基于表单的用户界面, 每个表单会调用一套内聚的、相互关联的应用功能。如果对每种类型的客户端请求都使用一个单独的控制器进行封装, 会导致控制器的实现复杂化, 而且如果对来自某个特定表单的请求的处理有通用的地方也会出现功能重复。

例如, 在使用静态HTML的Web应用中, 一些页面对特定数据记录的输入和处理的操作方式是, 取得输入数据记录, 如果该记录已经存在则从数据库中加载数据, 检查记录数据的完整性, 在应用中处理这些数据。与之类似, 数据处理的结果也要显示给用户。在执行安全、日志或其他后台功能的时候也会有很多通用的约束。如果严格地将页面上每个类型的功能封装到一个单独的控制器中, 忽视它们之间的相互依赖和通用的执行约束, 就会导致代码复杂化, 并引入冗余代码, 使得代码难以维护和改进。

因此, 为应用的用户界面中每个表单引入一个页面控制器以控制该表单上发出的所有请求的执行 (见图14-5)。

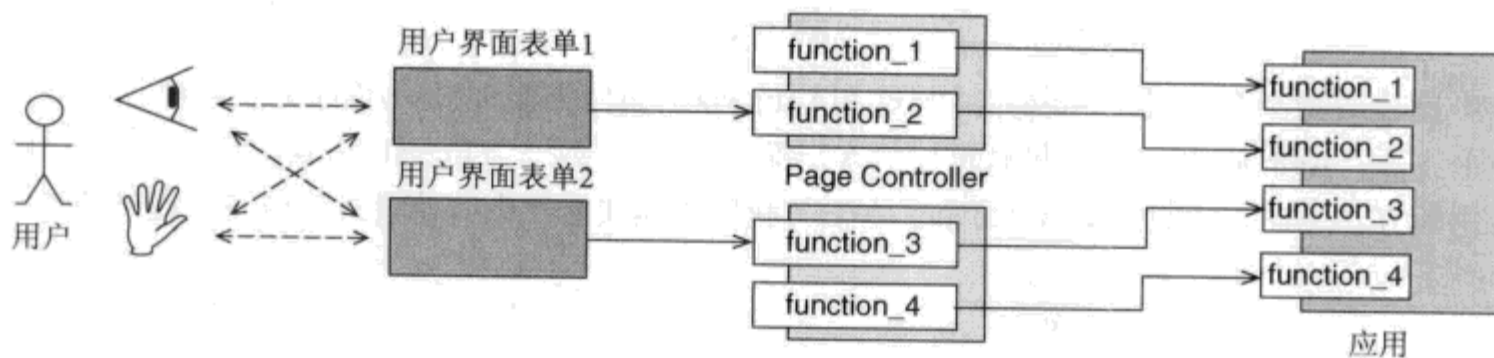


图 14-5

来自一个表单的所有的客户端请求通过与之关联的页面控制器传递给应用。页面控制器通过从客户端请求参数中提取消息等方法将每个请求转换成对应用组件的具体请求。页面控制器还负责与从该表单发出的所有与请求相关的辅助功能。而且, 如果表单中包含了一些表单内部的小型工作流, 页面控制器也可以用来维护必要的会话状态信息。



Page Controller将从某个特定用户界面表单中发起的对应用的领域服务的访问集中在一起, 这样通过这个表单发出单独的客户端请求时, 相关的功能上就不会出现重复了。通用的请求处理代码就更容易维护和改进。而且, 由于对不同表单的请求处理被分离开来, 这样就可以支持基于表单的请求处理策略, 或者在另一个用户层次的工作流内重新调整表单。

如果某些请求处理功能使用同样的核心流程, 只是具体策略上有些差别, 我们就可以使用Template Method (265) 或者Strategy (266) 来实现它。页面控制器的职能是将传入的请求转换成应

用组件上的服务请求。这些请求通常使用Command (240) 来实现, 以便能够实现对应用逻辑上请求的执行的执行的管理, 以及调度、日志、撤销/重做等特性。

页面控制器的部署方式大体上可以分为两种: 针对客户端的和针对应用的。针对客户端的部署方式可伸缩性更好, 但是要求客户端有专门的资源进行处理。与之相反, 针对应用的部署方式降低了对客户端资源的要求。如果多个客户端共享一个页面控制器, 通常我们需要对其进行同步。Thread-Safe Interface (224) 是一个简单的、粗粒度的方式, 因为即使方法中只有一小部分属于关键段, 它也要对页面控制器的接口进行同步。如果需要更精细的串行化, 也可以使用Strategized Locking (226) 完成同步。

14.2 Front Controller**

在开发Model-View-Controller (109) 架构的时候, 如果视图相对于模型是远程的……对于发送给某个应用的服务请求的处理和执行, 我们往往需要一种机制能够把它们联合起来。



网络应用在处理请求的时候经常执行一些相似的操作, 包括在执行动作之前或者之后进行授权和记录日志, 根据当前的情况为特定的用户显示特定的视图。如果在应用的每个控制器中各自实现一套这些功能就会出现很多重复代码, 从而使得代码难以维护和改进。

随着控制器越来越多, 从中找出哪些代码重复了会越来越难。于是对散落在各处的重复代码的修改便为错误或者不一致的应用行为敞开了大门。而且, 随着请求由笼统到具体的转换越来越复杂, 应用——尤其是Application Controller——的内存消耗会急剧增长。而且如果授权、日志这些通用的基础设施功能在每个请求中都需要执行的话, 情况就会变得更加糟糕。

因此, 引入一个Front Controller, 专门负责发布应用的功能, 并将客户端的服务请求转换成具体的可以在应用组件上调用的请求 (见图14-6)。

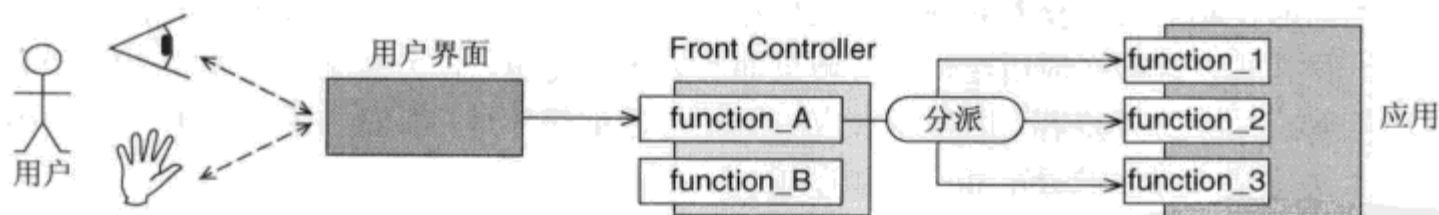


图 14-6

所有的客户端请求都通过Front Controller进行分派。Front Controller通过从客户端请求参数中提取信息等方式将请求转换成应用组件上的具体请求。



Front Controller将把一个应用的领域服务的访问集中在一起, 避免了与处理单个客户端请求相关联的功能的实现出现重复。这样的好处是, 不仅通用的请求处理代码更容易维护和改进, 也降低了应用的内存占用。而且Front Controller还可以提供授权和日志这些常用的辅助功能。但是如果很多个客户端的请求同时到达, Front Controller这种集中式设计可能会造成性能和扩展的瓶颈, 这是Front Controller的主要缺点。同时, 它也可能成为应用中的单点失败的罪魁祸首。

一般的请求转换和辅助功能可以用Template Method (265) 或者Strategy (266) 来实现。如果某一项辅助特性或者转换特性完全是可选的,也可以选择使用Decorator——有时也称为Intercepting Filter[ACM01]——来实现。经过Front Controller转换的结果是对应用组件的具体的服务请求。这些请求通常使用Command (240) 来实现,以便能够实现对应用逻辑上请求的执行的管理,以及调度、日志、撤销/重做等特性。

页面控制器的部署方式大体上可以分为两种:针对客户端的和针对应用的。针对客户端的部署方式会降低Front Controller性能、伸缩性和故障等缺陷的影响,但是会消耗客户端更多的内存和CPU资源。与之相反,针对应用的部署方式减少了对客户端资源的需求,但是前面所述的缺陷所带来的风险会更大。

如果Front Controller是由多个客户端共享的,通常我们需要对其进行同步。Thread-Safe Interface (224) 是一种粗粒度的同步方式,它对Front Controller的接口执行强制同步。如果Front Controller的方法中只有一小部分是关键段,也可以使用Strategized Locking (226) 来完成同步,后者提供了更为精细的串行化。Command (240) 对象是针对每个请求的,不存在共享的问题,因此它们的接口不需要做线程安全处理。

14.3 Application Controller**

在开发Model-View-Controller (109) 架构的时候,如果视图相对于模型是远程的……为了能够处理用户界面导航和应用的工作流,我们必须提供一个访问点。



应用往往是让用户根据某个工作流访问一系列的页面或者表单,或者是在某个条件下能看到某些页面或者表单。然而,如果将这些逻辑放到应用的控制器中,就会将用户界面的代码与专属于某个应用的工作流逻辑混在一起。

而且,不同的控制器可能会发起同一个工作流,这也会导致代码重复,从而增加维护和改进的难度。另一种方式是直接在应用逻辑中实现页面和表单的逻辑,作为对某个动作的响应。但是这个方法在实际应用中并不推荐,因为通常应用组件与显示部分是相互独立的,这样一来应用组件就会依赖于特定用户界面的分割和页面顺序了。

因此,将应用的工作流单独封装到一个Application Controller中。用户界面使用该Application Controller来决定调用应用逻辑中哪个动作,以及动作执行完之后显示哪个视图(见图14-7)。

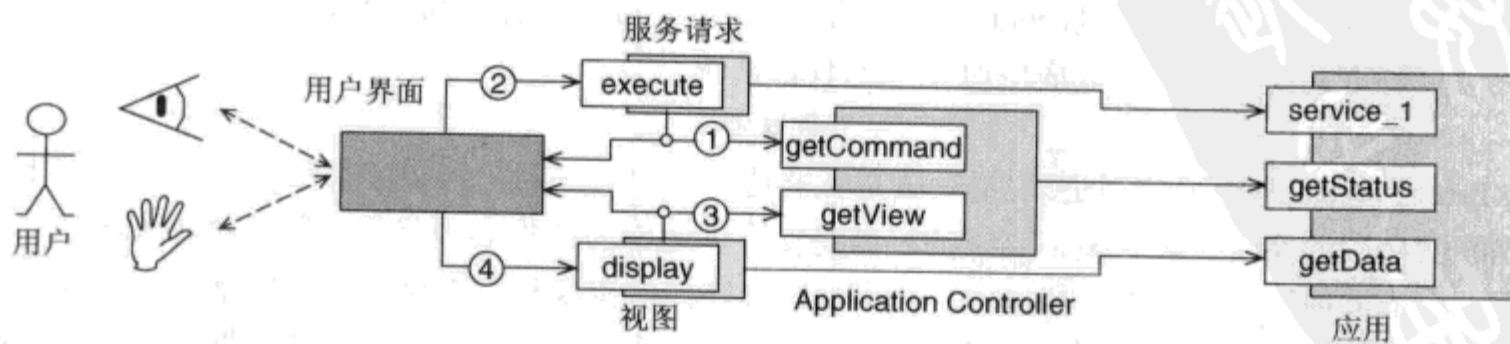


图 14-7

Application Controller为用户界面元素提供了一个集中的访问点,将对应用功能及其工作流的访问统一在一起。在用户界面控制器调用Application Controller的时候,后者负责找到应当执行哪项功能。选择哪项功能与从控制器收到的输入以及Application Controller当前的工作流状态都有关系。同样,在调用的功能执行完成之后,工作流进入到新的状态,作为对该功能的响应,Application Controller决定在用户界面中显示哪个视图。



Application Controller将应用中用户界面与领域逻辑交织在一起的部分封装起来。这样做的好处是应用的核心领域逻辑独立于包括用户界面结构在内的所有方面,同时用户界面控制器及其视图也独立于工作流和应用逻辑的状态。如果应用的工作流或者用户界面的结构发生变化,对应用代码的修改将主要集中在Application Controller中。在分布式应用中,Application Controller设计的这种依赖性,因其在应用中的这种中心角色可能会成为潜在的性能和伸缩性的瓶颈。同时它也可能成为单点失败的罪魁祸首。

通常,Application Controller将从用户界面控制器收到的请求转换成具体的Command (240) 对象。Command支持对应用逻辑上请求的执行进行管理,同时还支持调度、日志、撤销/重做等功能。

Application Controller的部署方式大体上可以分为两种。针对客户端的部署方式在上面提到的性能、伸缩性和单点失败等方面的代价要小一些,但是对客户端资源的要求比较高。与之相反,针对应用的部署方式减少了对客户端资源的需求,但是前面提到的缺陷会彰显出来。如果多个客户端共享一个Application Controller,我们必须对其进行同步。Thread-Safe Interface (224) 是一种粗粒度的方式,它对Application Controller的接口执行强制同步。如果Front Controller的方法中只有一小部分是关键段,也可以使用Strategized Locking (226) 来完成同步,后者提供了更为精细的串行化。

14.4 Command Processor**

在开发Model-View-Controller (109) 架构或者Active Object (212) 的时候,或者使用Command (240) 或者Message (245) 封装应用的服务请求的时候……我们需要一种执行服务请求的机制。



如果一个应用可以从多个客户端接收请求,我们就需要一种机制来管理这些请求的执行,包括处理请求调度、日志、撤销/重做等。通常,独立的客户端不会知道其服务请求应该在何时或者何种条件下执行。

允许客户端从应用中获得这些信息会增加其逻辑复杂性和物理复杂性,并且降低其模块化程度,影响到扩展性和可理解性。因此,客户端在发出请求的时候应当不需要知道自己的请求在什么条件下执行。然而,更糟的是很多分布式应用需要支持额外的辅助功能和系统管理功能,比如日志、授权和多次撤销/重做。这些功能既不当是应用客户端的事,也不当是应用组件的责任。如果让客户端负责这些事情,会增加客户端对组件的耦合程度,减少其内聚性;如果让应用组件负责这些事情,会让这些额外的功能与组件的本意混合在一起,从而增加进一步开发的难度。

因此,引入一个Command Processor,专门负责执行服务请求。Command Processor在应

用的约束下代表客户端执行请求（见图14-8）。

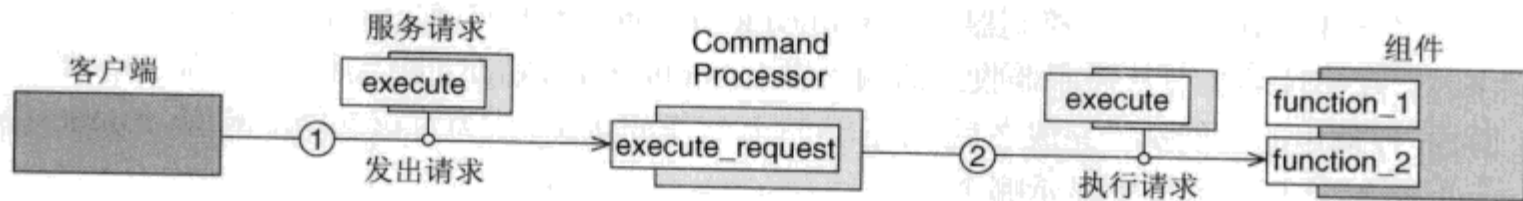


图 14-8

客户端只能通过Command Processor向应用发出请求。而且Command Processor可以访问组件的计算状态信息，因此它可以根据优先级、吞吐量等条件来调度挂起的服务请求。此外，Command Processor也可执行一些辅助功能或者系统管理功能，比如持久化或者那些依赖于过去某些状态的功能。



由于存在一个专门管理组件功能的Command Processor，组件的客户端和组件本身便不需要关心如何组织具体服务请求的执行了，从而降低了两者之间的耦合程度。

由于调用组件上服务的职能转移到了Command Processor，这意味着客户端就不能够直接调用组件上的功能了。客户端需要将“对象化的”请求——比如Command (240) 或者Message (245)——发送给Command Processor，由后者来执行组件上的请求。这里请求的具体化是使用Command Processor显式地、集中地处理请求的关键所在。

在Command Processor的内部，它可以通过多个Collections for States (276) 来接受请求。例如，可以用一个“do”集合来保存所有待执行的请求和被撤销后可以重做的请求，用一个“undo”集合来保存组件上所有执行过的请求，以便将来进行回滚。如果Command Processor收到的请求本身就是一个Command对象，它可以简单地调用这个对象来执行组件上封装好的请求。如果收到的请求是一个Message，则可以使用Interpreter (258) 将其内容转换成Command。

调度和其他的辅助功能——比如日志和授权——可以通过Template Method (265)、Strategy (266) 或者Interceptor (260) 来实现或者进行配置。通过这些模式我们可以使用不同的执行策略来对Command Processor进行配置，对绑定时间和耦合程度做出权衡和取舍。如果Command Processor由多个客户端共享，可以使用Monitor Object (214) 来实现。Strategized Locking (226) 为我们提供了在另一个维度上进行同步的方法。

14.5 Template View**

在实现Model-View-Controller (109) 架构的时候……我们往往需要使用某种预先指定的视图格式来展示应用数据和或者其他信息。



我们经常需要在应用中使用各种视图来展示动态的数据——比如数据库查询结果，这些视图每一次显示或者更新其内容都会发生变化。要想为每种可能的变化做一个单独的视图实现，不但会使得代码无限膨胀，也会引入大量的重复代码。

我们知道静态的视图，其内容、结构和外观都是固定的，因而易于开发。理想情况下，变化的视图其设计和实现也应当差不多简单。然而，视图的变化却意味着我们需要有不同的实现。我们能够看到，视图并不是任意地变化，所以我们应当找到一种方式使用单一的实现来处理这种有穷的变化。

因此，引入一个Template View，在这个Template View中预先定义好视图的结构，并为动态的数据设置好占位符（见图14-9）。

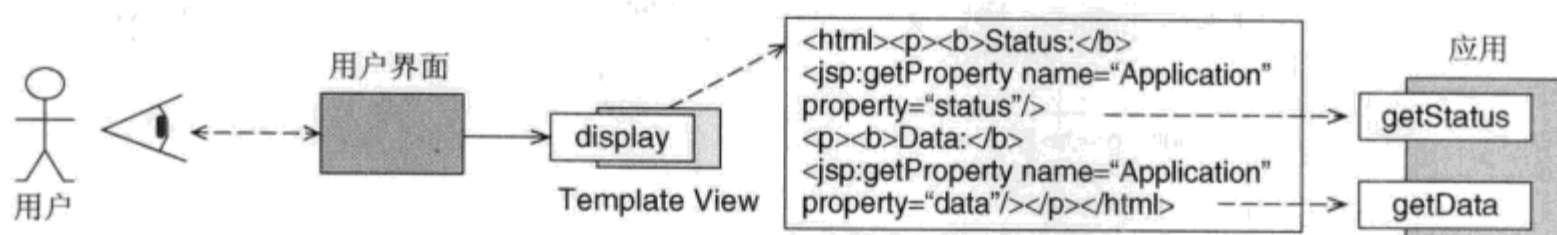


图 14-9

在显示或者更新这个视图的时候，我们使用相应的数据来填充这些占位符，以显示计算结果或者数据库查询结果等。在要显示的应用数据确定之后就可以将视图显示给用户了。



如果某些视图其结构相同，而只是内容发生变化，我们可以使用Template View来简化我们的开发。Template View使用一个单独的类将视图的结构、访问和计算内容的代码，以及在适当的位置显示这些内容的代码封装了起来。

Template View的常见的形式是服务器端页面，比如ASP.NET、JSP和基于PHP的页面生成。这些技术实际上已经不是纯粹的Template View了，因为你可以在里面嵌入任意的活动内容，包括编程逻辑。如果使用不慎，你可能就会在Template View中放置过多的应用逻辑，这与我们前面说的视图仅用来显示信息的目标其实是背道而驰的。这个问题可以通过在视图和应用组件之间插入一个helper类来解决，将必要的编程逻辑移到这个类里面[Fow03a]。如果在Template View中包含了根据某个条件显示内容或者迭代逻辑，也会遇到同样的问题。虽然这些情况无法完全避免，但是还是少用为妙，而且最好把逻辑提取到与该视图相关联的helper类中[Fow03a]。

14.6 Transform View**

在实现Model-View-Controller (109) 架构的时候……我们经常需要把从应用返回的请求响应数据转换成适当的视图。



应用一般都是通过多个视图将由应用服务返回的内容展现给用户，这些内容往往来自不同的、复杂的数据结构。然而，通常这些数据结构中并不包含关于哪些内容字段应当显示或者应当以什么样的形式显示的信息。

在视图内部实现对数据的检测，并将其转换成具体的输出格式，会导致应用逻辑和用户界面代码混在一起，所以这并不是一个好办法。这会使得我们很难将应用逻辑和用户界面清楚地分离开。而且，数据检测和转换的代码越大、越复杂，视图就越不适合低配置的瘦客户端，而这恰恰

是Web应用的典型的需求。

因此，引入一个Transform View，通过它遍历从应用中返回的数据结构，找到需要显示的数据，并将其转换成合适的输出格式（见图14-10）。

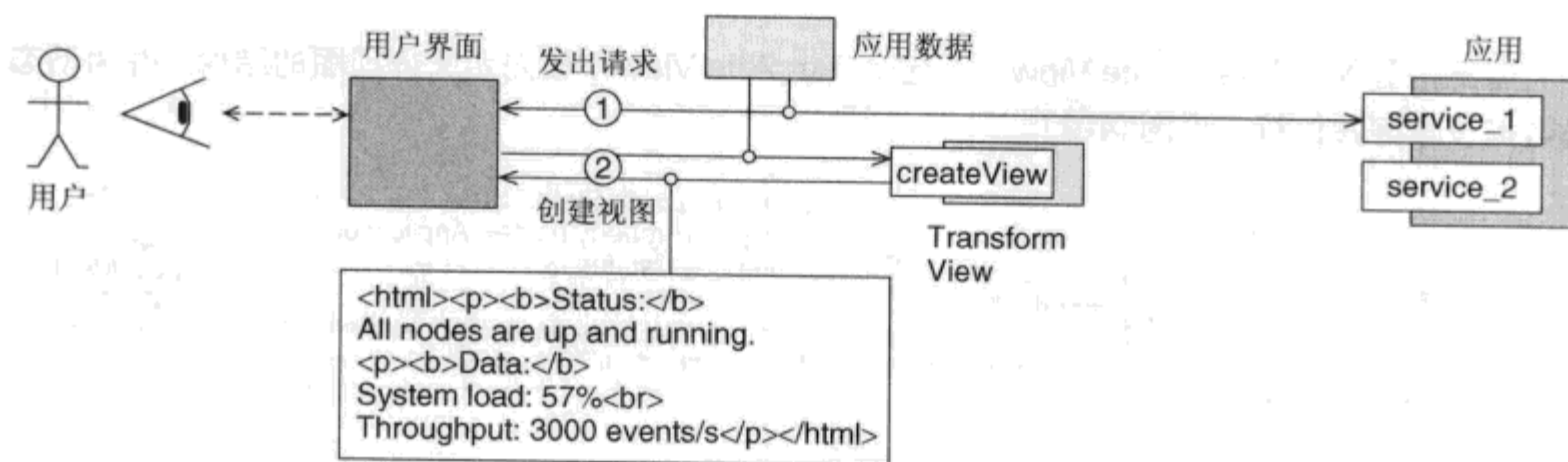


图 14-10

Transform View是一个适配器，它将一种类型的输入转换成另一种类型的输出。控制器从应用收到对服务请求的响应数据，将其传递给Transform View，然后接收到具体视图并将结果显示出来。



如果视图必须由复杂的数据集装配而成，使用Transform View可以简化视图的开发。应用中关于数据检测的逻辑不再与特定的用户界面技术和跟视图相关的数据呈现代码耦合在一起，同时也使得两者可以独立地变化。Transform View也可以作为可插拔的组件存在，从而为应用提供高度的灵活配置。

通常我们使用Data Transfer Object (244) 来封装Transform View所转换的应用数据，目的是保持Transform View和具体的数据结构之间的独立性。为了简化转换功能的编程工作，data transfer object可以将其内容序列化成XML等标准的展现形式。在这种设计中也可以使用XSLT等流行的数据转换技术。

Transform View的缺点是转换逻辑本身就是代码，而且是元代码，它缺少必要的工具支持，而Template View在这一点上则通常都有一些比较直接的解决方案。因此，它给人的感觉不是非用不可，再加上测试和调试转换的难度，很多开发人员都对其敬而远之。

14.7 Firewall Proxy**

在为可以通过互联网或者其他的公共网络访问的应用设计基于Proxy (169) 界面时……我们必须保护组件免受外来的攻击。



如果一个应用可以通过互联网等公共网络访问，通常它并不知道谁在使用它，所以从本质上讲是任何人都可以调用其功能。因此我们需要保护这些应用，以防止来自客户端的服务请求中嵌入攻击代码。

要保护可以公开访问的应用免受攻击并不是一件容易的事情。一方面这些应用必须可以公开地访问，所以我们就不能够保证应用预先知道哪些用户可能会访问它。至少我们要保证非注册的用户也可以访问主页、注册页面和其他的基本查询功能。另一方面，既然用户可以公开地访问应用服务，攻击者就可以在请求中嵌入攻击代码。根据应用的安全策略，我们需要把攻击识别出来并阻止其对应用的访问。

因此，为应用中可以公开访问的功能引入一个Firewall Proxy。通过这个Proxy为每个客户端请求添加相应的安全策略，从而保护了实现该功能的组件免受攻击（见图14-11）。



图 14-11

外部客户端只能通过Firewall Proxy访问应用的功能。客户端向Proxy发出请求，Proxy根据相应的访问规则检查该请求，比如检查请求消息中的内容。如果这个请求可以接受，Firewall Proxy就会把它传递给受保护的组件，否则该客户端请求就会被拒绝。



Firewall Proxy可以在数据包和请求内容这个层次上对组件的服务请求进行详细地过滤，这使得它可以在相对于网络协议做更高层次的攻击识别。而且安全检查和组件是分离开的，应用内部的客户端和防火墙后面的客户端都可以直接地访问组件，其性能也就不会受到安全检查的影响。

Firewall Proxy所应用的安全策略通常是基于一定规则的，而这些规则和Proxy本身是分开的。这样就使得我们可以单独地修改安全策略，而不会影响到Proxy的实现。跟其他的基于拒绝服务的安全机制一样，Firewall Proxy对于识别为恶意访问的客户端也会拒绝其对应用功能的访问，这样就存在错误识别的风险。因此Firewall Proxy的识别精确度严重地依赖于其基本规则的质量。

如果我们使用Packet Filter Firewall[SFHBS06]来实现Firewall Proxy，就可以将来自可信的客户端的请求直接转交给组件，而不需要检查请求的内容。使用Stateful Firewall[SFHBS06]来实现Firewall Proxy，可以避免对已经建立并且检查过的连接发来了请求执行检查。这两种设计都可以提高外部客户端的服务质量。将Firewall Proxy配置成Reverse Protection Proxy[SFHBS06]，对外部的客户端而言又添加了一层安全屏障，不论请求由谁发送、通过哪个连接收到或是请求的内容有多么干净，受保护组件的可访问功能都会受到限制。

最后，如果公开的功能是由部署到多个服务器的多个组件实现，Firewall Proxy也可以用Integration Reverse Proxy[SFHBS06]来实现。这样外部客户端和潜在的攻击者就无法了解到部署的模型和网络的拓扑结构。

14.8 Authorization**

在实现Client Proxy (139) 或者Explicit Interface (163) 的时候……我们必须确保只有特定的客户端可以访问某个子系统的功能。



子系统必须为其客户端提供定义良好的和有意义的功能, 否则它对客户端来说就没有什么价值了。客户端可以通过发送服务请求调用这些功能。然而, 并不是所有的可以向子系统发送请求的客户端都有权调用这些功能。

维护敏感信息——比如医疗和金融数据——的子系统不能被未授权的客户端访问。而且也不是所有的用户或者子系统的内部客户端都可以访问或者管理这些信息, 否则这些数据的机密性、完整性及其可用性就会受到损害。那些负责管理和配置任务的组件也有同样的问题。直接在组件中对可信的客户端信息进行硬编码的解决方案, 不仅不够优雅而且有很高的维护成本, 因为要对多个应用和不同的客户端重新部署组件, 或者更换授权的客户端, 或者改变客户端的授权都需要巨大的精力和成本。

因此, 如果一个子系统本身是对安全敏感的, 在客户端向其发送服务请求的时候, 为客户端分配合适的访问权限, 并在子系统执行每个请求之前检查这些权限 (见图14-12)。

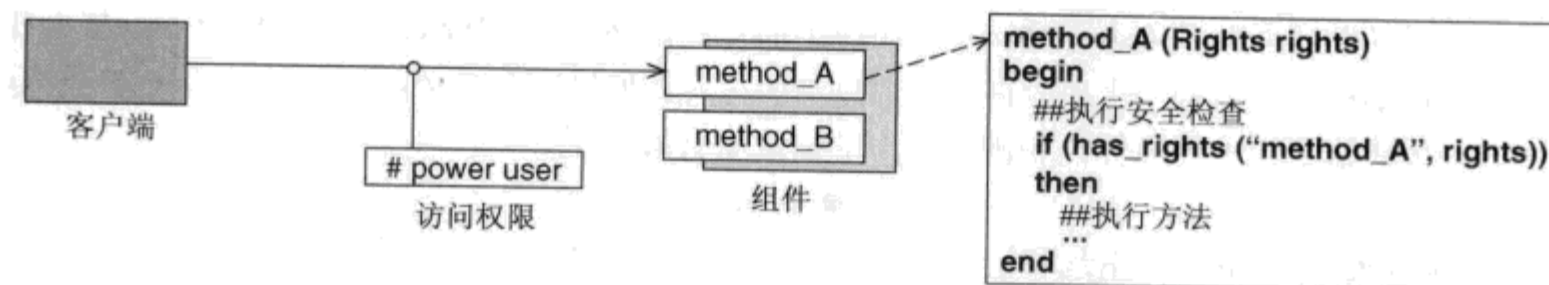


图 14-12

访问权限指定了客户端可以在子系统上调用哪些或者哪类操作。任何一个客户端向子系统发送服务请求, 我们都会首先检查它的访问权限。如果它们有足够的权限, 相应的服务请求就可以执行, 否则访问就会被拒绝。

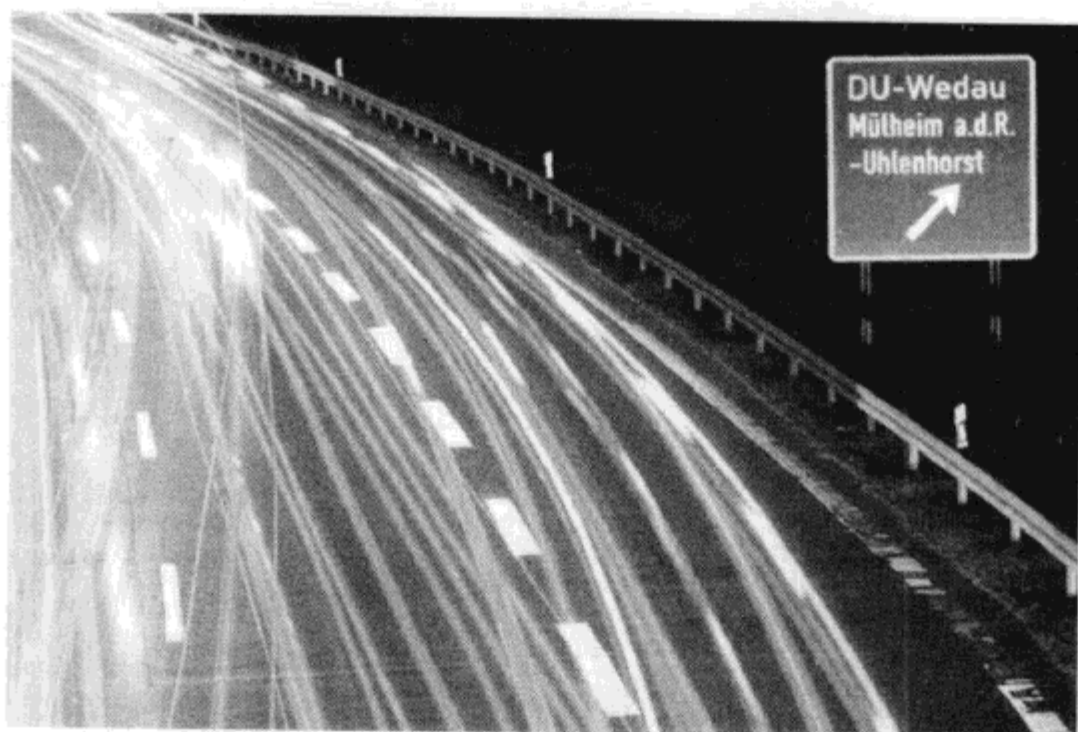


Authorization最大的好处是子系统只能被授权的客户端访问, 这对于安全敏感的应用来说是非常重要的。第二个好处是访问权限和检查策略可以透明地改进, 不会对应用的客户端和组件产生任何影响。

Authorization基础设施的常见的实现形式是Role-Based Access Control[SFHBS06]。在应用中预先定义好一套用户角色, 比如管理员、高级用户、用户、访客, 将客户端和一组访问权限相关联, 而不是将每个独立的访问权限直接分配给每个客户端。用户和客户端组件必须属于某个或某几个角色。根据其角色访问权限的不同, 它们可以——或者不可以——访问应用的组件及其功能。Role-Based Access Control通过将大量的访问权限组织成少数几个角色, 从而简化了Authorization的实现和改进。而且Role-Based Access Control设计可以推行拥有该应用的组织的结构和策略。

Reference Monitor [SFHBS06]可以为Authorization结构提供定义好的访问权限决策和强制实施点。所有的客户端请求由一个集中的引用监测器来检查其访问权限是否符合应用的授权规则。这个设计通过检查所有的客户端服务请求严格地推行应用的安全策略。引用监测器通过将组件的访问权限检查集中在一起，避免了代码重复，简化了维护工作。可以在一个系统中部署多个引用监测器的实例以避免出现性能和伸缩性的瓶颈或者单点失败。





A3高速公路杜伊斯堡-韦道出口夜景
©西门子公司 西门子出版社

选择什么样的并发架构会对多线程软件的设计和性能产生极大的影响，而对分布式软件的影响尤为明显。然而，还没有一种并发架构是适合所有的负载情况和平台的。本章所介绍的4种并发模式可以应对多种并发问题——从将异步并发处理与同步并发处理组合起来，到将对共享组件的访问进行同步，同时保证性能和吞吐量的最大化。

分布式系统软件通常能够从并发中获益，尤其是处理从多个客户端同时发出请求的服务器和服务器端软件。同时，人们设计出越来越多的多核CPU和多CPU计算机来运行多控制线程，以弥补相对于摩尔定律的差距[Sut05a]。因此，进程和线程管理机制成了分布式系统软件开发人员必须精通的知识。

进程是一组资源的集合，比如虚拟内存、I/O句柄、控制线程，它为执行程序指令提供上下文。在硬件保护地址空间（hardware-protected address space）中，每个进程便是一个保护和资源分配单元。相反，线程是一个独立的指令序列，它以进程作为运行的上下文。线程不仅包含指令

的指针，同时还包括诸如函数激活记录（function activation records，即调用栈）、寄存器组以及线程专属（thread-specific）的数据等资源。一个线程是一个执行单元，它属于某个进程，并且与进程中其他的线程共享地址空间。

促使分布式系统软件使用多进程和多线程的原因是多方面的，包括下面这些。

- 通过使用现代硬件和软件平台的进程并发能力透明地提高性能。
- 允许程序员进行交叠计算和在服务处理过程中交互，从而明显地提高性能。
- 对于交互式——比如包含用户界面的——软件可以缩短感知响应时间，因为不同的线程执行不同的服务处理任务，用户可以在某些任务阻塞的时候做其他的工作。
- 允许多个服务处理任务独立运行，采用同步的编程抽象——比如双向方法调用，以及阻塞I/O和锁操作——从而简化应用设计。

然而，编写高效、可预料、可缩放而且健壮的并发软件是相当困难的[Lea99]。高效的并发编程绝不仅仅是把独立的组件、对象或者服务用各自的控制线程启动起来，然后就可以撒手不管了。究其原因，包括以下几点。

- 软件的多样性。既然不同类型的分布式系统软件的结构和行为特点各不相同，所以也就不存在放之四海皆适用的并发模型。例如，有些软件混合使用异步和同步服务处理，有些软件则由事件驱动，还有的软件必须处理不同优先级的服务请求。因此，每个类型的软件所要求的并发模型都可能有所不同，以便为用户提供有质量保证的服务，同时为开发人员提供合适的编程模型。
- 多线程成本。并发软件的设计者必须清楚多线程会引入上下文切换、同步和在CPU缓存间移动数据的开销。轻率地使用线程机制很可能会减少从并发中获得的好处，甚至得不偿失。因此，设计并发软件时应该尽量将应用多线程的开销降到最低。
- 可移植性。已有的软件开发方法、工具和操作系统平台的局限也会给并发编程带来额外的复杂性。例如，现代硬件和软件平台的多样性使得开发能够运行在多种操作系统上的并发软件和工具变得异常复杂。

有效地解决这些挑战和复杂性要求开发人员了解并能够正确地运用并发模式。在设计软件的基线架构、子系统和组件的整个过程中，都应该自觉地、认真地理解和应用这些模式。

我们的分布式计算模式语言包含了4个模式，实践表明这些模式可以帮助我们创建多种并发架构和解决各种设计问题。它们如下所示。

- Half-Sync/Half-Async（半同步/半异步）模式(209)[POSA2]对并发系统中的异步和同步服务处理解耦合，以简化编程，而不会过度地影响性能。该模式引入两个相互通信的层，一个用于异步服务处理，另一个用于同步服务处理。
- Leader/Followers（领导者/跟随者）模式(211)[POSA2]提供了一个高效的并发模型，在该模型中多个线程轮流使用一套事件源，来检测、分离、分派和处理事件源中出现的请求。
- Active Object（活动对象）模式(212)[POSA2]通过将服务请求和服务执行解耦合来提高并发性，它将对象化的服务请求放到自己的控制线程中，并简化了对它们的访问。
- Monitor Object（监控对象）模式(214)[POSA2]通过同步并发方法的执行来保证同一时刻

一个对象中只有一个方法在运行。它允许一个对象的多个方法以协作的方式确定它们的执行顺序表。

本章的范围仅限于开发分布式系统并发通信的中间件和应用程序组件。我们的目的并非要覆盖与并发相关的所有方面。我们把主要的关注点放在几个关键的模式上面。这些模式定义了如何构造 (structure) 和分割 (partition) 并发软件, 从而形成多个协作的线程, 以及如何组织对由多个线程共享的组件的访问。如今在理论上和实践上已经存在很多成功的并发模型, 但是这里我们有意没有做笼统的、指导性的展示, 因为这并非本章的目的所在。

有关线程同步技术的模式也没有包括在本章, 而是在第16章中。虽然 Thread-Specific Storage 模式在 [POSA2] 中归类为并发模式, 多年的使用经验告诉我们它更多的是关于如何避免加锁成本——而不是并发, 因此我们把它也归到了同步这一章。

Monitor Object 归于并发模式还是同步模式也有些争议。我们把它放到了这一章, 因为其主要目的是在面向对象编程中应用并发, 它也使用了互斥、条件变量等同步机制, 但是相对来说并非其主要特点。值得指出的是 Monitor Object 是 Active Object 的补充。

本章中的模式分为两组: 并发基础设施和访问同步。

Half-Sync/Half-Async 和 Leader/Followers 定义了高层并发架构。Half-Sync/Half-Async 对并发系统中的异步和同步处理耦合, 以简化编程, 而不会过度影响操作系统和网络层次的性能。由于其编程模型简单, Half-Sync/Half-Async 被用于多种并发应用, 包括操作系统、中间件、工业流程控制和通信应用 [POSA2]。Leader/Followers 为事件驱动系统提供了并发模型。它尤其适用于处理短暂的 (short-duration)、原子的并且反复的动作中的事件, 比如接收和分发网络事件或者向数据库中存储大量数据记录。

图15-1展示了 Half-Sync/Half-Async 和 Leader/Followers 是如何与我们的模式语言集成在一起并应用于分布式计算的。

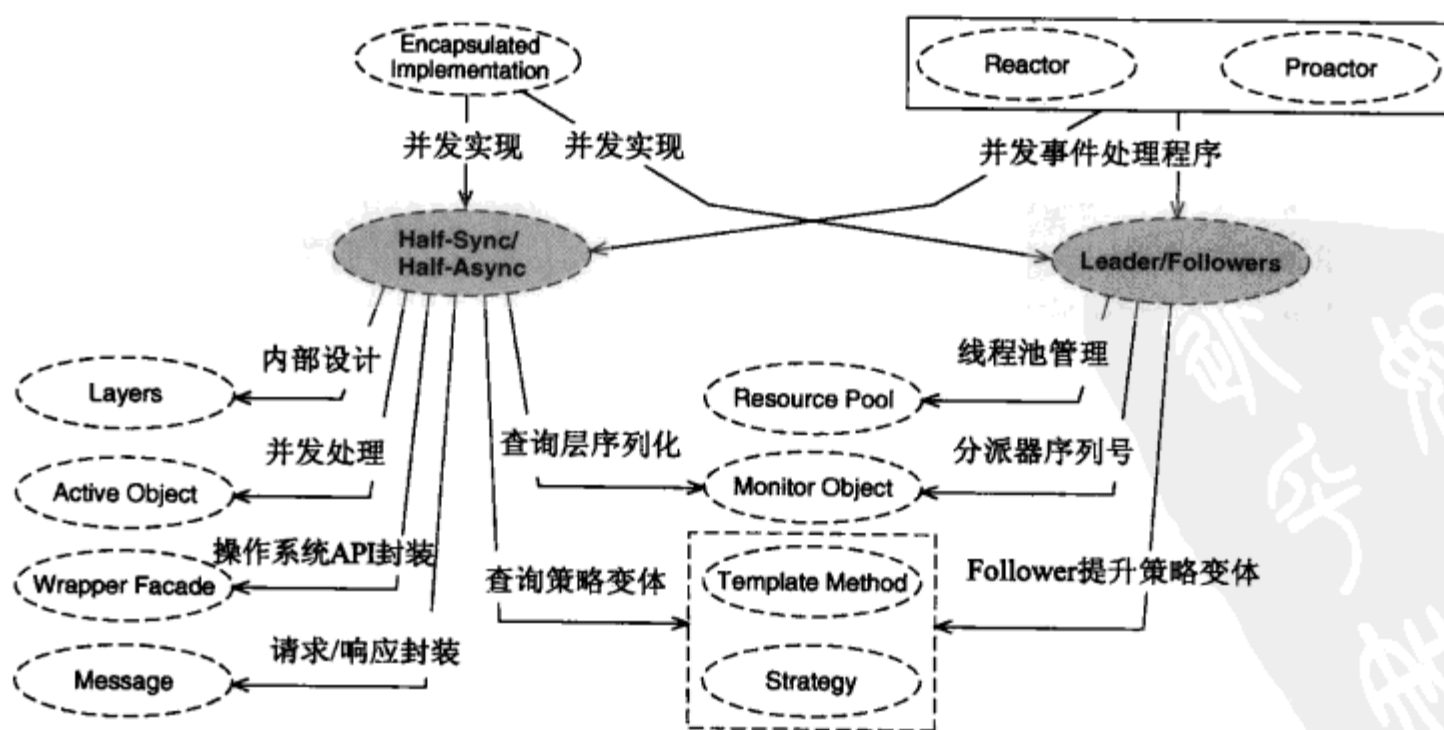


图 15-1

Active Object和Monitor Object模式可以对对象和组件上并发调用的函数进行同步和调度。其主要区别是Active Object的方法是在与其客户端不同的线程中执行，而Monitor Object的方法则在其客户端线程中执行。所以Active Object可以执行更为复杂——尽管代价很大——的调度，来确定其方法的执行顺序。Active Object主要用于在大的组件和子系统中支持并发，而Monitor Object则主要用于实现较小的并发对象。

图15-2展示了Active Object和Monitor Object是如何与我们的模式语言集成在一起并应用于分布式计算的。

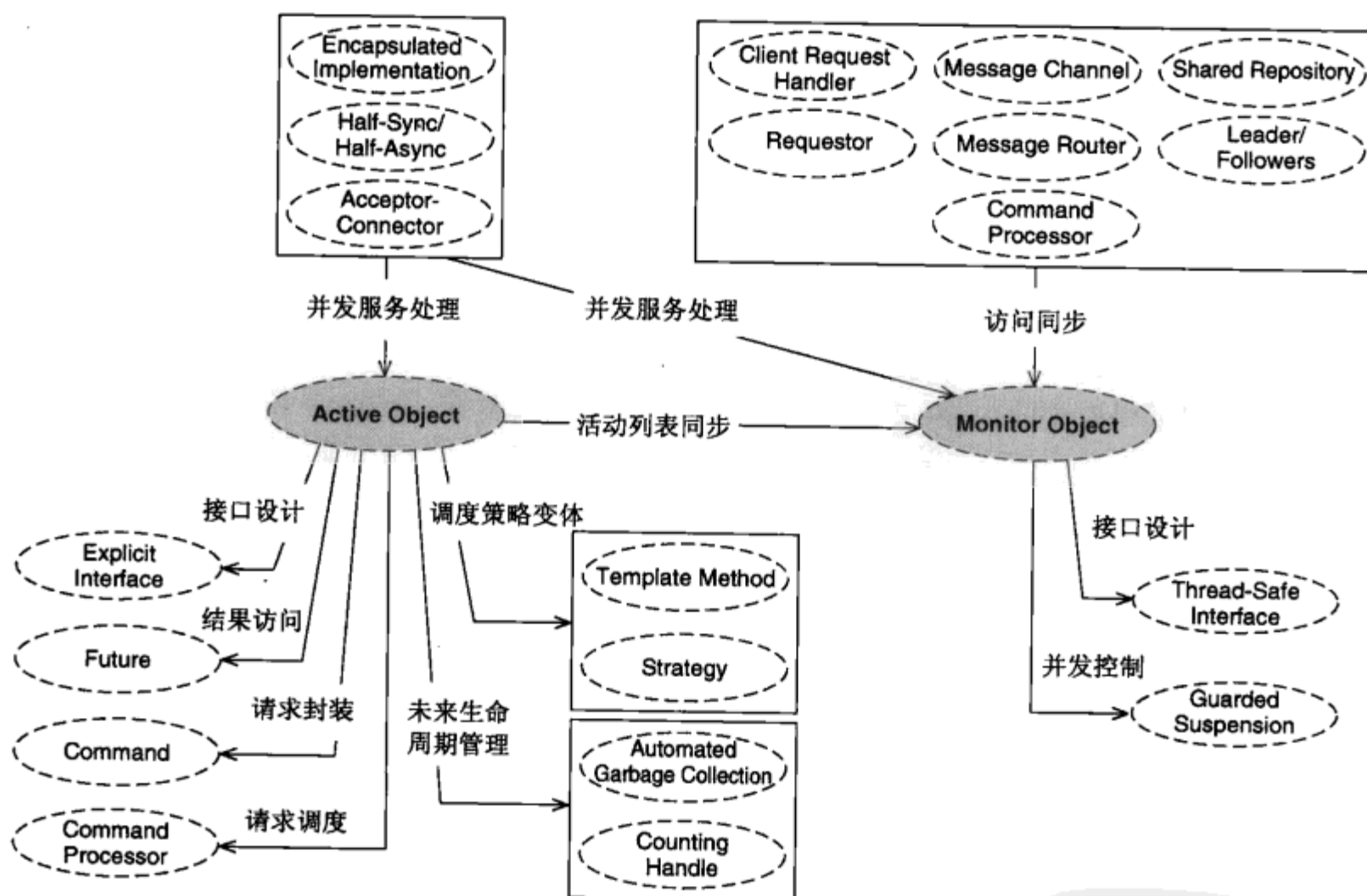


图 15-2

15.1 Half-Sync/Half-Async**

在开发并发软件，尤其是并发Encapsulated Implementation (181) 或者应用了Reactor (150) 或 Proactor (152) 作为事件处理基础设施的网络服务器时……我们需要在确保对并发机制的应用能够简化编程的同时，还要保证性能上的高效和可伸缩性。



并发软件中的服务处理通常是异步和同步同时存在。异步用于高效地处理底层系统服务，同步则用于简化应用服务处理。要想从两种编程模型中均获益，高效地协调异步和同步服务处理是

非常重要的。

异步和同步服务处理通常是相关联的。例如，Web服务器的I/O层往往使用异步读操作来取得HTTP GET请求[HP97]。而在CGI层对GET请求的处理则同步地运行于独立的控制线程。在I/O层异步到达的请求必须与CGI层对请求的同步处理集成在一起。换一个角度，我们再来看看Web客户端，AJAX可以使用异步JavaScript和XML来提高Web客户端的可感知响应速度[Gar05]。通常，异步和同步服务应该相互协作、取长补短。

因此，将并发软件的服务分解为异步和同步两层，然后增加一个队列层来处理二者直接的交互（见图15-3）。

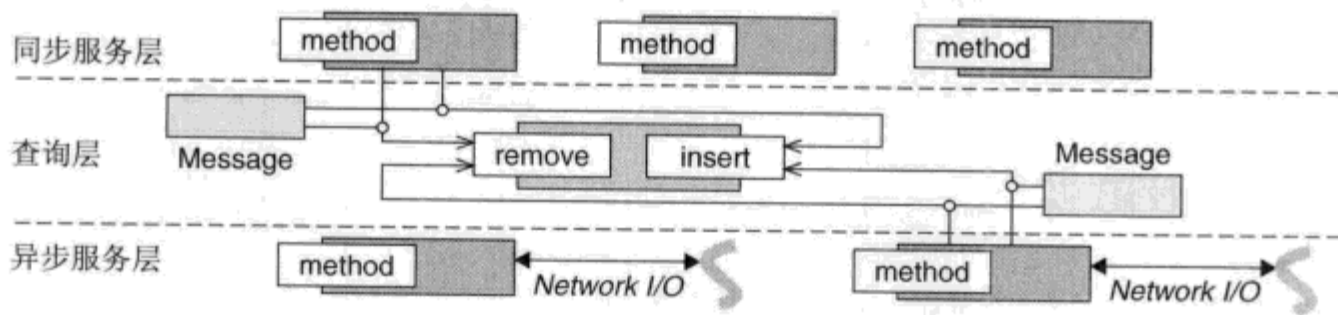


图 15-3

使用单独的线程或者进程以同步的方式处理高层服务，比如领域功能、数据查询或文件传输，而由网络硬件中断所驱动的短暂（short-lived）协议处理程序等底层系统服务则应该采用异步方式。如果同步层的服务必须和异步层的服务通信，则应当采用排队层交换消息的方式进行。



Half-Sync/Half-Async模式对这3层进行了严格的划分，这使得并发软件更容易理解、调试和改进。而且，异步和同步服务各自的缺陷也不会出现相互传染：异步服务的性能不会因为同步服务被阻塞而降低，同步服务的编程简单性也不会受到异步复杂性（比如显式的状态管理）的影响。最后，使用排队层可以避免对异步和同步服务层之间的依赖进行硬编码，同时也可以简化消息处理的优先级排序。异步层和同步层的严格解耦合要求这两层之间的通信必须或者使用Copied Values——这样会带来性能上的损失和资源管理上的消耗，因为有更多的数据需要传送；或者使用Immutable Values表示——这种做法相对来说属于轻量级的方式，但是其构造上可能更为复杂。

总的来说，Half-Sync/Half-Async通过使用Layers (108) 来保证3个不同的执行模型（model）和通信模型的独立性和封装性。

同步层服务，比如数据库查询、文件传输或者领域功能通常运行在自己的线程中，以便多个服务可以同时执行。如果一个服务实现为Active Object (212)，它便可以同时处理多个服务请求，从而提高应用的性能和吞吐量。

异步层的服务可以通过异步中断或者支持异步I/O的操作系统API实现，后者包括Windows重叠I/O（Windows overlapped I/O）和I/O完成端口（I/O completion ports）[Sol98]，或者POSIX异步I/O aio_*族系统调用[POSIX95]。Wrapper Facade (269) 用于将与平台相关的异步I/O功能封装成统一的接口，藉此简化异步层的正确使用和跨平台移植。如果我们将Half-Sync/Half-Async设计成与Proactor或者Reactor事件处理基础设施联合使用，这个事件处理基础设施便是所谓的异步层。虽

然Reactor并非真正的异步，但如果它的服务实现的是短时（short-duration）操作，而不是长时间的阻塞，你就会发现它其实具有异步的关键属性。

排队层通常是由同步层和异步层所有服务共享的消息队列组成。复杂的排队层可以提供多个消息队列，比如为每个消息优先级或者通信端提供一个消息队列。消息队列可以实现为Monitor Object (214)，这样就可以使得异步和同步服务能够对消息队列进行透明的串行访问。Template Method (265) 和Strategy (266) 用于支持对消息队列不同方面的设置，例如配置其排序、串行化、通知和流程控制的行为。Strategy相对更为灵活，它可以为消息队列提供更松的耦合度，并支持运行时配置和重配置。Template Method则更适合于仅需要编译期灵活性的情况。由排队层进行路由的信息封装在Message (245) 里面。

15.2 Leader/Followers**

在开发并发软件，尤其是并发Encapsulated Implementation (181) 或者应用了Reactor (150) 作为事件处理基础设施的网络服务器时……我们常常需要并行而高效地响应和处理从多个事件源发起的不同事件。



大多数事件驱动的软件使用多线程来并行地处理多个事件。然而，要想以高效的、可预见的和简单的方式为线程分配工作却出奇地困难。

在事件驱动软件中，尤其是服务器软件中，经常需要在线程和事件源之间定义高效的事件分离机制。同时，在使用多线程分离一系列共享事件源上的事件的时候，还需要防止出现竞争状态。例如，一个Web服务器可能在多个I/O句柄上使用多个线程服务于多个GET请求，以保证其可扩展性。有些将线程和事件源关联起来的方法效率太低，而且需要付出高昂的代价。例如，为每个请求创建一个线程，或者为每个事件源安排一个单独的线程，由于操作系统和硬件的扩展性限制，这些方式的效率都非常低。所以，我们需要一个既容易实现、效率又高的并发响应软件架构。

因此，使用一个预分配的线程池来协调事件的检测、分离、分发和处理。在这个线程池中，一次只有一个线程——Leader（领导者）——等待一系列共享的事件源上的事件。当事件到来时，Leader将池中另一个线程提升为新的Leader，然后自己与其他处理线程以并行的方式进行事件处理（见图15-4）。



图 15-4

在Leader监听事件源等待事件出现的同时，其他线程（即Follower，跟随者）处于排队和休眠状态，直到被提升为Leader。如果当前的Leader线程检测到事件源中的一个事件，它会做两件事情：首先，它把一个Follower线程提升为一个新的Leader，然后它自己将变为一个加工者（processor）

线程，将事件分离并分派给指定的事件处理程序，事件处理程序运行在接收事件的线程中。在当前的Leader线程等待共享事件源上发生新的事件的同时，多个处理线程可以并行地处理事件。处理完事件，处理线程还原为Follower角色，并且保持休眠状态直到再次被提升为Leader。



通过预先分配一个线程池，Leader/Followers设计避免了动态线程创建和销毁的额外开销。将线程放在一个自组织的池中，而且无需交换数据，这种方式将上下文切换、同步、数据移动和动态内存管理的开销都降到了最低。而且，让Leader线程执行对下一个Follower的提升可以避免由于存在一个中心提升决策者而带来的性能瓶颈。

而为这些性能优化所付出的代价是其受限的适用范围。Leader/Followers仅在处理短暂的、原子的、反复的和基于事件的动作上取得了成功，比如接收和分发网络事件或者向数据库存储大量数据记录。事件处理程序所提供的服务越多，其体积也就越大，而处理一个请求所需的时间越长，池中的线程占用的资源也就越多，同时也需要更多的线程。相应地，应用程序中其他功能可用的资源也就越少，从而影响到应用程序的总体性能、吞吐量、可扩展性和可用性。

在大多数Leader/Followers设计中共享的事件源封装在一个分配器组件中。如果在一个设计中联合使用了Leader/Followers和Reactor事件处理基础设施，其响应（reactor）组件便是分配器。封装事件源将事件分离和分派机制与事件处理程序隔离开来。如果正在提升新的Leader线程，而对最近的事件的处理恰好同时完成就可能出现竞争状态，为分配器提供停用和重激活特定事件源的方法，可以避免这种情况。

线程池可以用Resource Pool (298) 来实现，而事件的分配和对共享事件源的同步访问则可以用Monitor Object (214) 来实现。该设计通过使用自组织的并发模型来提高性能，该模型可以避免在事件源和事件处理程序中间引入单独的查询层的开销。

在线程池中，Monitor Object为线程提供两个方法。一个是join（加入）方法，使用这个方法可以把新初始化的线程加入到池中。新加入的线程将自己的执行挂起到线程池监听者条件（monitor condition）上，并开始等待被提升为新的Leader。在它变成一个Leader之后，它便可以访问共享的事件源，等待执行下一个到来的事件。另一个是promote_new_leader方法，当前的Leader线程使用这个方法可以提升新的Leader，其做法是通过线程池监听者条件通知休眠的Follower。收到通知的Follower继续执行（resume）线程池的join方法，访问共享事件源，并等待下一个事件的到来。

我们还可以通过使用Template Method (265) 和Strategy (266) 来支持多种提升协议，比如后进先出、先进先出和最高优先级等。

15.3 Active Object**

在开发Encapsulated Implementation (181)，或者Half-Sync/Half-Async (209) 架构中的同步服务层，或者Acceptor-Connector (154) 配置中的服务处理器时……我们通常需要确保组件的操作可以在它们自己的控制线程中并行地运行。



并发通过允许组件同时处理多个客户端请求而不必阻塞来提高软件服务质量。然而，如何在软件中表现并发单元，以及如何在运行中与之交互，则是由开发人员决定的。

客户端尤其应该能够随时向组件发出请求，而不必等着其他请求执行完毕。我们需要能够根据请求的优先级或者截止时间等特征调度客户端请求的执行。为了保证服务请求的独立性，它们的串行化和调度对组件和客户端都应该是透明的，以便在对软件实现进行重用的时候不必考虑它们是否要求不同的同步策略。

因此，将并发单元界定为组件上的服务请求，并且让它和请求的客户端在不同的线程上运行。使客户端和组件可以异步地交互来产生和使用服务结果（见图15-5）。

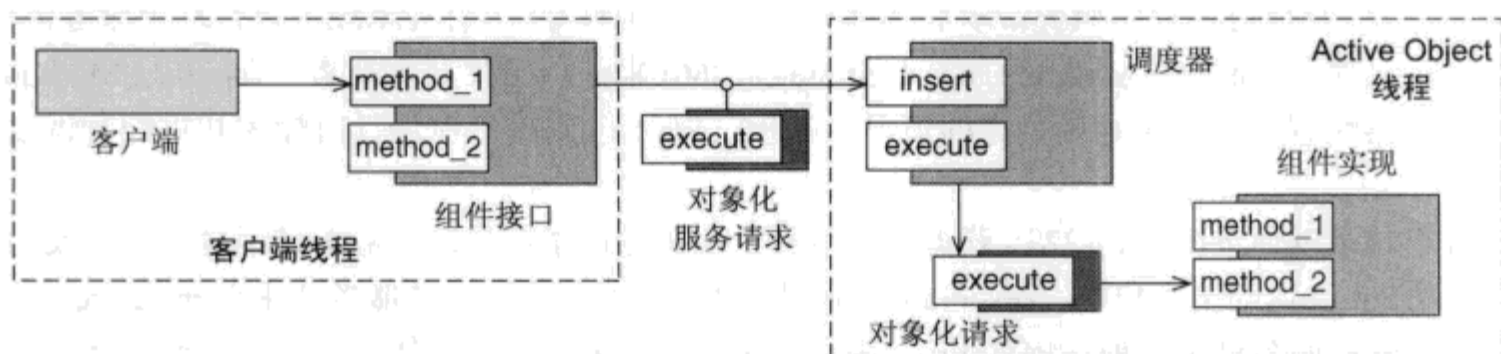


图 15-5

客户端可以通过调用组件暴露给客户端线程接口上的方法初始化组件上的服务请求。设计组件接口的时候不必受同步的约束，在发出请求之后立刻把控制返回给客户端。将请求对象化，并传递给运行在一个或多个单独线程上的组件实现，让组件实现独立地调度服务请求的执行，而不依赖于何时对请求对象进行的初始化。此外，在服务结束后还要为组件提供将结果返回给客户的机制。



Active Object设计通过允许客户端线程和服务请求同时运行来增强应用程序中的并发性。在服务请求执行的过程中客户端并不会被阻塞。而且，通过使用调度器减少了同步的复杂性，调度器可以根据一定的同步约束机制来达到对组件实现的串行化访问。从实际组件实现中将需求调度分离出来，这样我们的组件就可以在不需要同步的情形中重用。同时有些组件可能并不是为并发访问设计的，这种分离机制使它们可以在并发的应用中使用。最后，服务请求的执行顺序可以与服务请求的调用顺序不同，这样在采用优先级排序、截止时间和其他的同步约束时就更加方便了——不过这样做会使得调试更为复杂。Active Object也会引入比较重量级的请求处理和执行基础设施，这对那些只实现短暂方法的组件会导致性能上的损失。

使用Explicit Interface (163) 将组件的接口暴露给客户端线程。从客户端的角度来看就跟组件位于客户端线程内是一样的。设计接口的时候要确保方法签名中不包含同步参数。这样即使并发组件是由多个客户端线程共享的，从客户端看来就像自己拥有独占的访问一样。

在运行时，组件接口将所有的方法调用对象化为服务请求——这通常使用Command (240) 来实现——并使用它来完成相应的方法调用的同步约束机制。请求的对象化将服务请求和服务执行在时间和空间上进行了解耦合，这样当客户端调用组件上的服务时就不需要阻塞自己或其他的客

户端。将创建的服务请求存储到一个共享的活动列表中，该列表维护了并发组件上所有挂起的服务请求。这个活动列表用Monitor Object (214) 实现，可以确保并发访问是线程安全的。

组件的实现可以有一个或多个宿主机线程。在每个线程中，有一个实现组件的功能servant。调度器将服务请求对象从共享的活动列表中分离出来，然后在servant上执行。这样的设计允许服务请求和执行并行地运行，也就是说，服务请求在客户端线程中调用，而服务执行则运行在不同的线程中。而且，调度器将组件功能从调度和同步机制中分离出来，使得双方都可以独立地开发和改进。

将调度器设计为Command Processor (199) 可以实现组件的事件循环 (event loop)。它负责检查活动列表，发现可执行的服务请求后将其从活动列表中移除，并在servant上执行这个服务请求。我们还可以使用Template Method (265) 和Strategy (266) 来支持多种调度策略。Template Method适用于调度器配置可以在编译期确定的情况。相反，Strategy则支持在运行时配置和重配置调度策略。

客户端可以通过Future (223) 获得并发组件上的服务请求结果。在服务调用之后，并发组件的接口将Future返回给客户端，在servant完成服务执行之后，由关联的服务请求填充Future。如果客户端在获得服务结果之前访问Future，客户端可以阻塞或者轮询等待，直到服务结果有效。如果我们不再使用Future，可以通过Automated Garbage Collection (307) 安全地将其回收——如果编程语言支持的话，否则，如果必须手动编码回收可以使用的Counting Handle (309)。

15.4 Monitor Object**

在开发Shared Repository (117) 架构，Requestor (140)、Client request Handler (143)、Message Channel (130)、Message Router (134) 分布式基础设施，Encapsulated Implementation (181)，Acceptor-Connector (154) 结构，Half-Sync/Half-Async (209)、Leader/Followers (211) 或者Active Object (214) 并发模型的时候……我们必须考虑对象要能够被多个线程共享。



并发软件经常包含一些对象，其方法会被多个客户端线程调用。为了保护共享对象中的内部状态，有必要将客户端对它们的调用进行同步和调度。然而，为了简化编程，我们不希望客户端在编程的时候需要区分访问的组件是共享的还是非共享的。

每个会被多个客户端线程访问的对象应该确保其方法的序列化是透明的，不需要客户端做显式地干预。为了确保其客户端的服务质量，如果共享的对象中的某个方法在执行的过程中必须阻塞，它应该自动地放弃其控制线程，保证组件处于稳定状态，以便其他客户端线程可以安全地访问。

因此，执行共享对象（的方法）应该是在其客户端线程中进行，并让它能够自己协调这个串行化而又交错的执行序列。必须通过同步的方法对共享对象进行访问，确保同一时刻只能执行一个方法（见图15-6）。

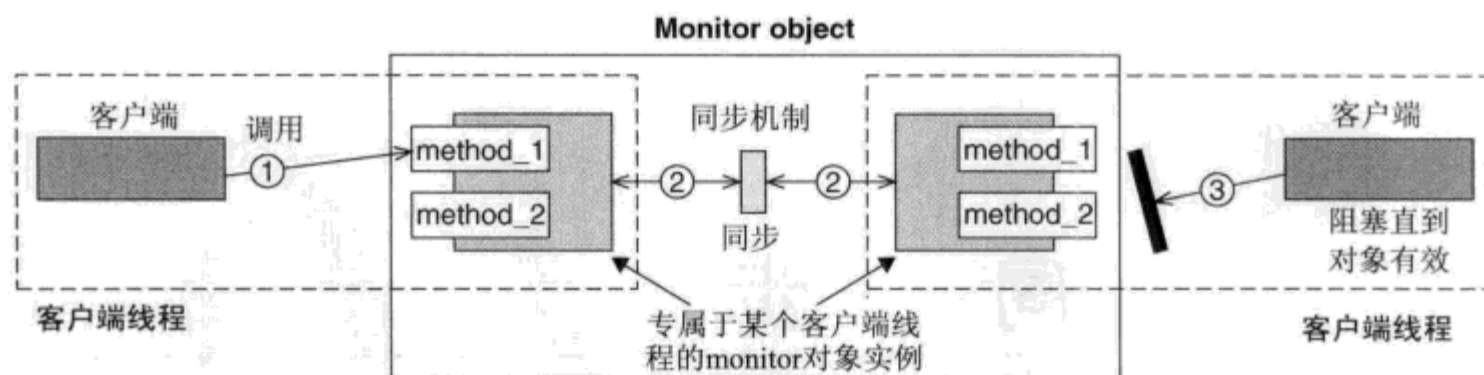


图 15-6

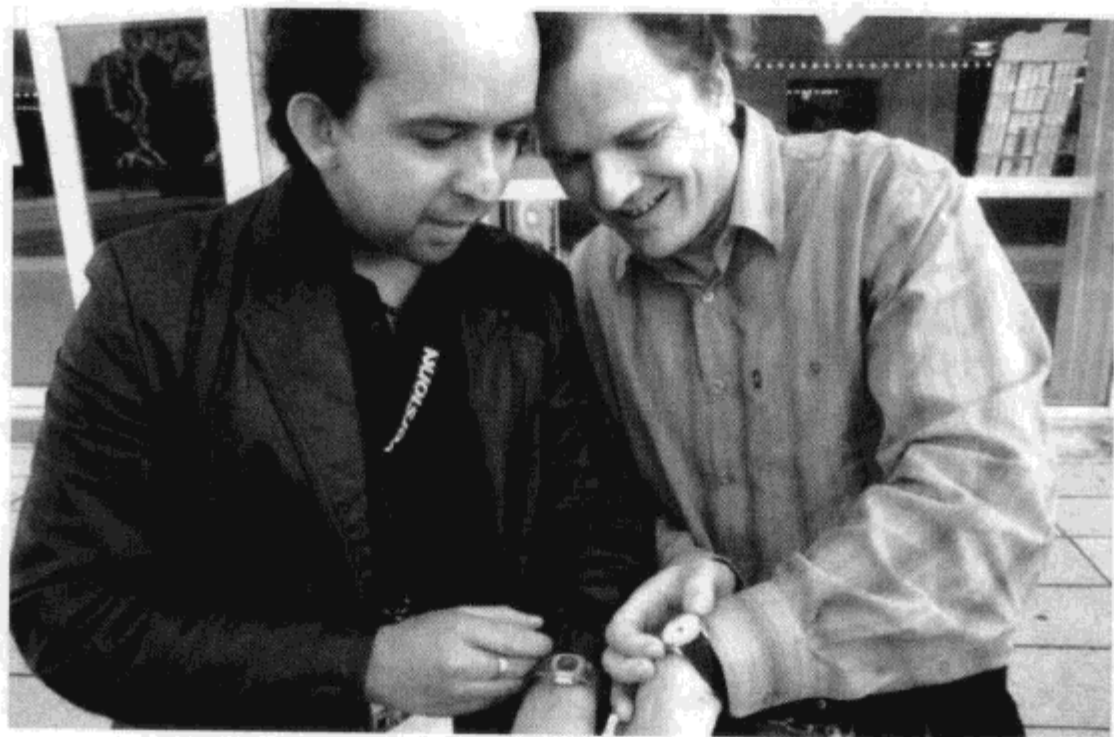
每个Monitor Object包含一个监听者锁，用于访问对象状态时串行化。在一个同步的方法中，首先获得监听者锁，确保没有其他串行化方法可以执行。获得这个锁之后，检查共享对象的当前状态是否允许运行串行化方法。如果允许，则执行它，否则在某个条件上挂起对同步方法的执行。如果该方法的执行挂起在某个监听者条件上，则监听者条件应该挂起调用者的线程，直到收到通知重新唤醒调用者线程。在线程挂起时，监听者条件应该释放监听者锁，并且在继续该线程的时候重新获取监听者锁。

挂起同步方法允许其他线程通过其他同步方法访问共享对象。任何同步方法在其执行结束时都可能会对监听者条件的有效性产生影响。这时应当通知相应的监听者条件，以便挂起的方法可以继续执行。在结束一个同步方法之前要释放监听者锁，以便其他线程调用的其他同步方法可以执行。



将共享对象的类型设计为Monitor Object——通过在协作的线程中共享对象，并将方法调用和状态同步结合起来——简化了并发控制。Monitor Object对于实现方法执行序列之间的协作也有帮助，确保共享对象对客户端是可用的，并在序列化条件约束下最大化共享对象的可用性，确保其状态的变化是完整的而且不会带来竞争状态。然而，这个模式的缺陷是它把领域功能和同步功能紧密地耦合在一起。编写或者使用Monitor Object经常会带来死锁的问题。例如，如果一个Monitor Object回调了另一个对象，而这个对象使用了其他的Monitor Object，就容易出现死锁。

Monitor Object可以使用Thread-Safe Interface (224) 来将同步行为和其他功能解耦合。这样两者就可以独立地变化了。Guarded Suspension (221) 可以用于协调对象中运行的线程。方法的执行是由监听者条件和监听者锁来调度的，它们决定它们（包括协作的组件）在什么情形下应当挂起，什么情况下应当继续执行。



Kevlin和Frank在2006年JAOO 会议上对表

©Mai Skou Nielsen

划分任务并在不同的线程上执行，这可不是并发的全部含义。当对象在多个线程中共享时，对其方法的使用会存在线程安全的问题。本章给出了9种模式以解决同步问题或减少状态的变化从而降低同步方面的需求。

在第15章并发中我们讨论了并发是如何帮助我们改善分布式系统的结构、提高系统性能和响应能力的。我们可以看到要设计一个结构好、效率高、响应快的并发软件有多么的困难，多线程这个东西就是说两个任务可能先后执行，也可能同时执行[MeAl04a]。为了应对这些挑战，我们在模式语言中包含了数个应用于高效并发设计的模式。但是对并发性系统来说不仅仅设计是有挑战性的，代码编写也是如此——一旦存在并发，我们在架构的各个层次和方面都要给予专门的考虑。

并发编程（Concurrent Programming）比顺序编程（Sequential Programming）困难的原因之一在于它对共享资源的访问需要同步。并行运行的线程可能会访问到同一个对象。有的方法需要修改共享对象的内部状态，如果不进行适当的保护，从不同的线程调用该方法时就可能破坏这个

状态。为了避免这个问题，那些不应当同时访问同一对象状态的代码应当被同步到一个临界区（Critical Section）中。临界区是满足下面规则的指令序列：如果某一个线程或进程在临界区中执行，则任何其他线程或进程都不能在同一临界区中执行[Tan95]^①。

在面向对象编程中保护临界区的通用方法是将类或组件与某种类型的锁对象相关联。例如，互斥对象（Mutual Exclusion Object，或称互斥体Mutex）是一种锁类型，它必须在同一线程内在进入和离开临界区时顺序地获取和释放，这样，如果多个线程试图同时获取互斥体，只有一个线程能成功。其他线程必须等到互斥体被释放，这之后等待线程才能再次竞争锁定互斥体[Tan92]。其他类型的锁机制，如文件锁（File Lock）、信号量（Semaphore）和读写锁（Reader-Writer Lock），都运用类似的获取-释放协议[Mck96]。

尽管在线程中使用锁从概念上来看很直观，在实际中用它来编程却并不简单[Lee06]。例如，如果保持线程锁比实际需要的时间长，使得锁定范围（在获取锁和释放锁之间的语句）比临界区（必须防止并发性访问的语句）大，会降低共享组件的可用性。在过细的粒度上获取和释放锁则会降低组件的性能——获取锁和释放锁的操作并不是免费的。

我们来看一下，在Java中锁定范围是怎样被轻易地扩大的。在Java中，锁定范围是一个显式概念，通过使用synchronized关键字引入来标记整个方法或一段代码。从语法上来说，标记整个方法为synchronized更方便，并且由于它的互斥操作是在方法签名上体现的，所以也更容易查看。然而，我们通常会在构成临界区的语句前后加上与本地变量或参数相关的语句，而不仅仅是修改私有对象状态，因此并不需要锁定机制。当必须采用锁机制时，我们需要确保程序的正确性；当不需要锁定时，应当尽量避免使用锁，以提升性能。在我们的例子中，整个方法的范围要比临界区的范围大，因此只有临界区需要被包含在synchronized块中，而不是整个函数。

草率地使用锁机制也会引入死锁的风险——两个并发性任务互相等待对方的完成而造成的死循环。考虑两个线程和两个共享资源的情况：线程1获取了第1个资源的锁，线程2获取了第2个资源的锁。为了进一步处理，线程1需要锁定并使用第2个共享资源，线程2需要锁定并使用第1个共享资源。因此，它们都无法继续下去，死锁导致了两个线程永久性挂起。设计策略和算法来确保总是按照同样的顺序获取和释放锁可以解决大多数问题，但是最保险的做法是减少同步操作和锁定的需要，并尽可能地将锁定行为封装起来[CMH83]。

采用锁定原语（locking primitives）的编程需要了解其基本的约束和局限以避免劣化（pessimization）和死锁。最简单的锁是信号量，它是一种能被任意线程锁定和解锁的互斥锁。更复杂一点的结构是互斥体，它的锁定和解锁必须在同一个线程内完成。这两种方式共同的问题就是容易忘记将锁定操作和相应的解锁操作对应起来。有时候这是明显的：你可能在源代码中直接漏掉了解锁的代码。但是也有时候可能比较隐晦：解锁的操作存在，但是当临界区产生异常时根本就不会执行到解锁部分的代码。正是由于这种隐晦的情形，我们更加需要对这种原语使用进行包装，或者将其包装在库代码中，或者通过语言机制来包装。

^① 注意不要将“临界区”（Critical Section）的正式定义与Microsoft Windows同步原语的CRITICAL_SECTION相混淆。前者定义了必须满足特定规则的代码区域，而后者是一个可以被用于满足该规则的API机制。CRITICAL_SECTION是轻量级的进程互斥类型。

更加微妙的是，互斥体有两种基本的风格：递归的和非递归的。递归互斥体允许可重入的锁定，如果一个线程在已经锁定某个互斥体，那么它可以再次将其锁定，并继续执行。非递归互斥体正好相反：同一线程如果试图再次锁定同一个互斥体就会导致自死锁（self-deadlock）。非递归互斥体通常比递归互斥体在锁定和解锁时速度更快，但是会有自死锁的风险。在对象调用自身方法时（不管是直接调用还是回调）需要特别注意，因为两次锁定会导致线程挂起。可递归互斥体是专门为这种场景设计的，它在回调这样简单而常见的情形中不会导致死锁，可以简化组件的组装。

另一个常见的同步机制是条件变量，它用来协调线程暂时中止自己直到条件表达式包含的线程间共享数据到达期望的状态。条件变量总是与互斥体同时使用，线程必须在检验条件表达式之前获取互斥体，如果条件表达式为假，线程自动将自己挂起在条件变量上并释放互斥体，因此其他线程可以改变共享数据。当一个合作线程改变了这些数据时，它能通知条件变量，从而自动恢复先前挂起在该变量上的线程并再次获取互斥体。

我们对同步的兴趣通常关注于基于锁的编程，特别是怎样高效地使用锁，而不只是充分地使用。正确性、安全性和有效性是推动我们的技术词汇。然而，锁并不是编写线程安全代码的唯一途径。设计软件以减少状态改变的机会能降低执行同步操作的需要。如果多个线程工作在不相交的数据上，它们之间就不需要同步。工作在共享的不变数据上的线程之间同样也不需要同步，因为不会有变化发生。工作在可以原子更新的数据上的线程也能共享状态改变。最后一个方案激发了无锁编程，它是由特定原语操作支持的，例如整数递增或比较并交换（Compare-And-Swap, CAS）操作在给定平台上是原子的而不需要锁定。然而，无锁编程是很精妙而复杂的主题——“基于锁的编程专家都很难做对，而无锁编程天才也很难做对”[Sut05b]，它在我们给出的模式中并不构成一个关键的主题。

我们分布式运算模式语言中的9个模式能有助于确保线程间和状态间的交互不会产生竞争条件和死锁，同时仍然尽可能保证效率。

- ❑ Guarded Suspension（守护挂起）模式 (221) [Lea99] 协调客户端对共享对象的透明访问，使其方法只能在特定条件满足的时候执行。
- ❑ Future（未来）模式 (223) [Lea99] 提供一个“虚拟”的对象，当客户端在其他并发性计算尚未完成的情况下试图访问对象的字段时，它会自动将客户端阻塞。
- ❑ Thread-Safe Interface（线程安全接口）模式 (224) [POSA2] 可以降低锁定开销，并确保组件内方法调用不会因为要再次获取自己持有的非重入锁而导致“自死锁”（Self-deadlock）。
- ❑ Double-Checked Locking（双检锁）模式 (225) [POSA2] 对于临界区代码只需要在第一次遇到时执行而以后不再需要执行的情形下，可以降低竞争和同步开销，而且仍然保证线程安全性。
- ❑ Strategized Locking（策略锁）模式 (226) [POSA2] 通过组件参数化允许用户选择最合适的同步机制来完成对组件的临界区的串行化。
- ❑ Scoped Locking（区域锁）模式 (227) [POSA2] 确保控制在进入某一范围内时自动获取锁、离开该范围时自动释放锁，而不必关心是从什么路径离开该范围的。

- Thread-Specific Storage (线程专属存储) 模式 (228) [POSA2] 允许多个线程使用一个“逻辑上全局的”入口点以获取线程内的对象, 而不会对每次对象访问引入锁定开销。
- Copied Value (副本值) 模式 (230) [HenOOB] 确保值对象通过复制方式在线程间传递。因为值对象不会在多个线程间共享, 所以不存在数据竞争的机会, 也就不需要同步。
- Immutable Value (恒值) 模式 (231) [HenOOB] 在构造时设置值对象的内部状态, 而且不允许将来改变其状态。Immutable Value可以在并发性程序中共享而不需要同步。

当多个线程试图执行某个方法或者访问某个数据的时候, 如果某些条件未能满足或者所需的数据尚未准备好, 那么就可能无法成功执行或者访问。列表中的前两个模式—Guarded Suspension和Future就可以帮助我们对这些线程进行协调来解决这个问题。Guarded Suspension综合协调对一个或多个条件变量的有效使用以实现协作性的并发控制, 而Future模式暂时挂起对正在运算数据的访问直到数据可用。图16-1描绘了Guarded Suspension和Future模式是怎样集成到我们的分布式运算模式语言中的。

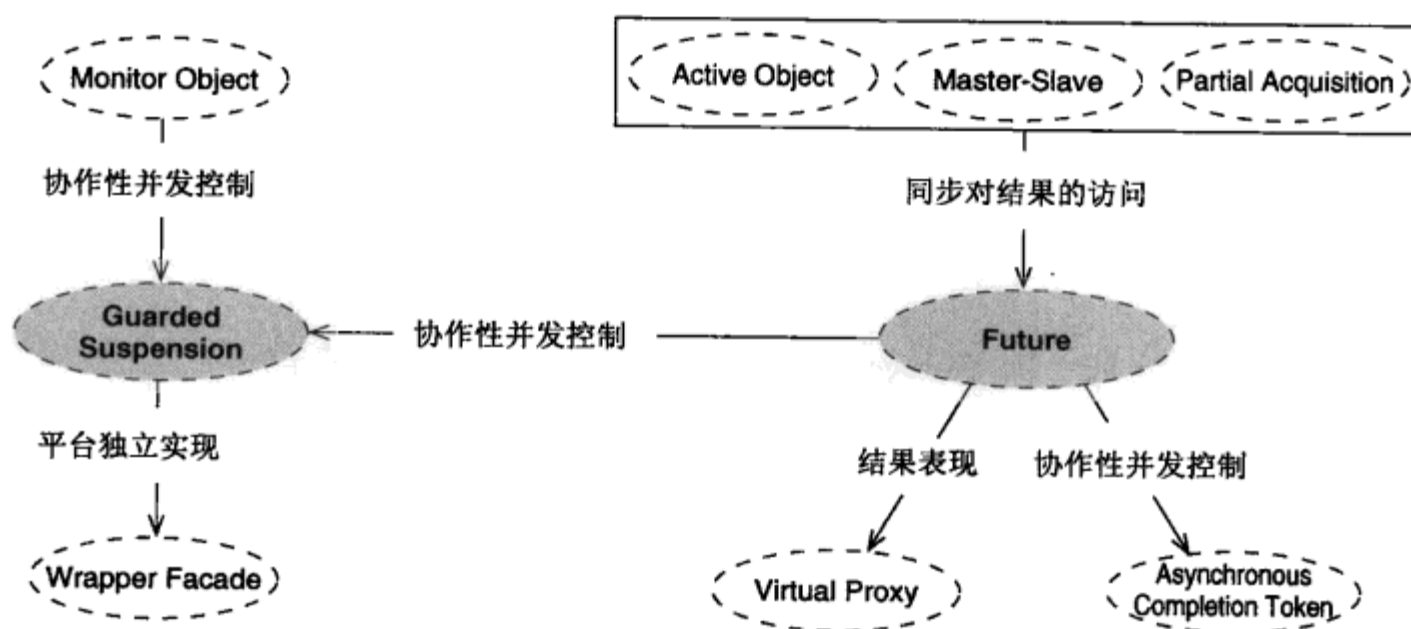


图 16-1

本章接下来给出的3个模式: Thread-Safe Interface、Stratigized Locking以及Scoped Locking给出了几个具体的串行化技术。Thread-Safe Interface实现了“粗粒度”的锁定策略: 它直接在组件接口上进行串行化, 而不考虑所调用的方法的全部处理时间有多少花在临界区域 (critical region) 内。花在“非临界区域”代码上的时间越多, Thread-Safe Interface的性能就越差, 因为它无谓地阻塞了别的线程。在这种情况下, Stratigized Locking和Scoped Locking是更好的选择, 因为它们直接在临界区域进行串行化。这两种模式还是互补的: Stratigized Locking定义了可插拔的锁类型, 可以根据应用的需要进行配置; Scoped Locking帮助我们实现自动获取和释放锁的操作, 后者往往更为重要, 因为这对非正常方式离开临界区域的情况特别有用, 例如在出现异常时。

图16-2描绘了这3种模式是怎样和我们模式语言中的其他模式联系起来的。

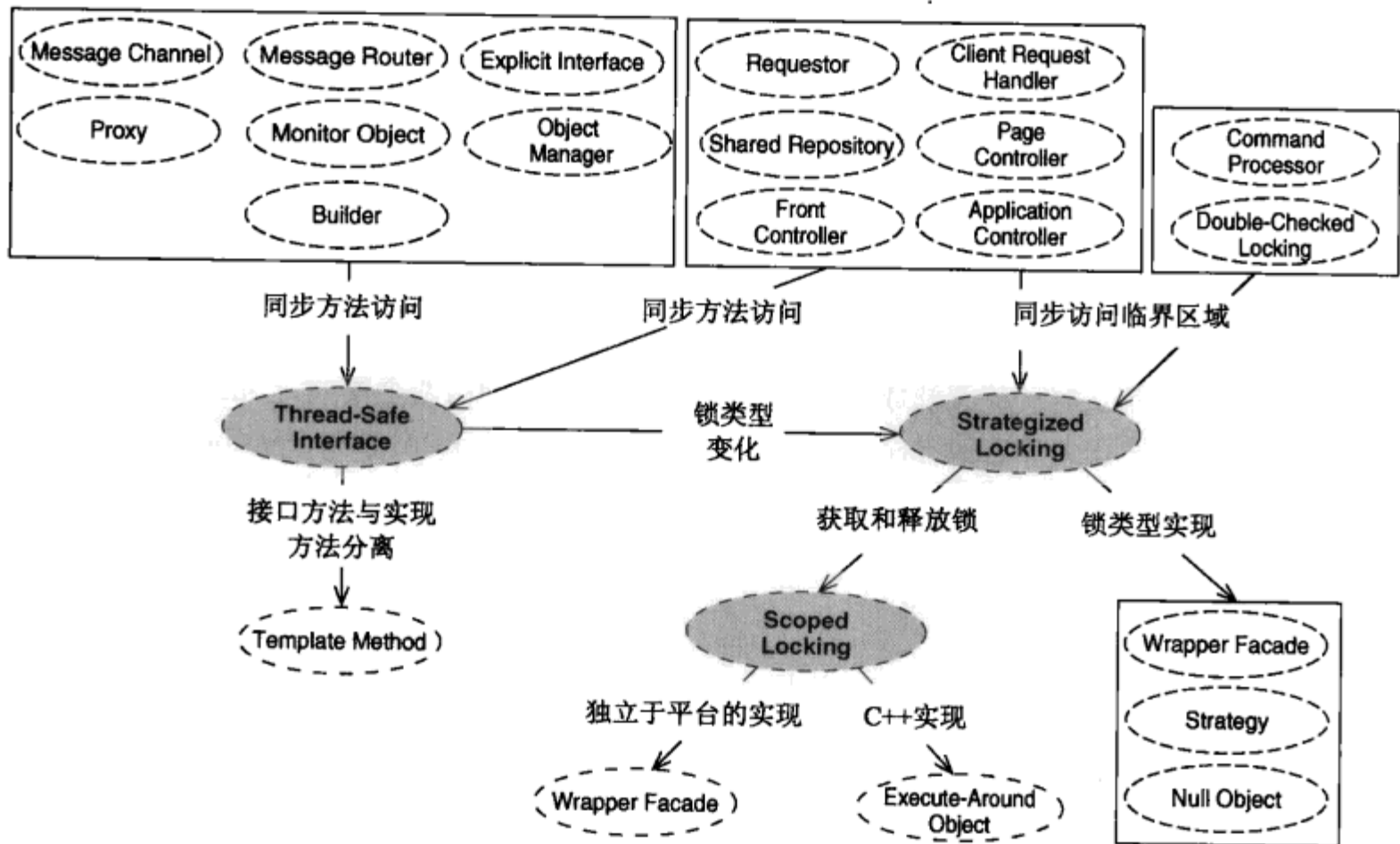


图 16-2

原来我们将 Thread-Specific Storage 模式划归为并发模式 [POSA2]，在模式语言中我们将其划归为同步模式。这是因为与 Double-Checked Locking 类似，Thread-Specific Storage 模式与并发性的关系较小，更多的是关于如何避免加锁的开销。如果要通过全局变量或者入口点（如 Unix `errno`）来访问线程局部变量，我们可以使用 Thread-Specific Storage 来避免锁开销。相反，如果临界区域是由某个条件表达式保护的，这个表达式的值依赖于某个状态，而这个状态会在临界区修改，这种锁机制就叫做 Double-Checked Locking。比如 Singleton [GOF95] 的初始化就是有了这种机制。

类似地，Copied Value 和 Immutable Value 彻底消除了加锁的必要性，从而避免了锁开销。Copied Value 通过复制对象来传递值，避免了共享，它可以确保多个线程看到的对象是互不相干的，也就不需要同步。一旦对象创建，不提供任何会引起内部状态变化的方法，这种限制使得 Immutable Value 从本质上就是线程安全的，是线程间交换数据最理想的方式。

对 POSA 系列第 2 卷 *Patterns for Concurrent and Networked Objects* [POSA2] 熟悉的读者可能会注意到我们将 Double-Checked Locking Optimization 模式更名为 Double-Checked Locking 以简化模式名。然而去掉“Optimization”的主要原因在于如果我们保留它，那么我们的模式语言中很多其他模式可能也需要加上这一属性。

图 16-3 描绘了 Double-Checked Locking、Thread-Specific Storage、Immutable Value 和 Copied Value 是怎样集成到我们的模式语言中的。

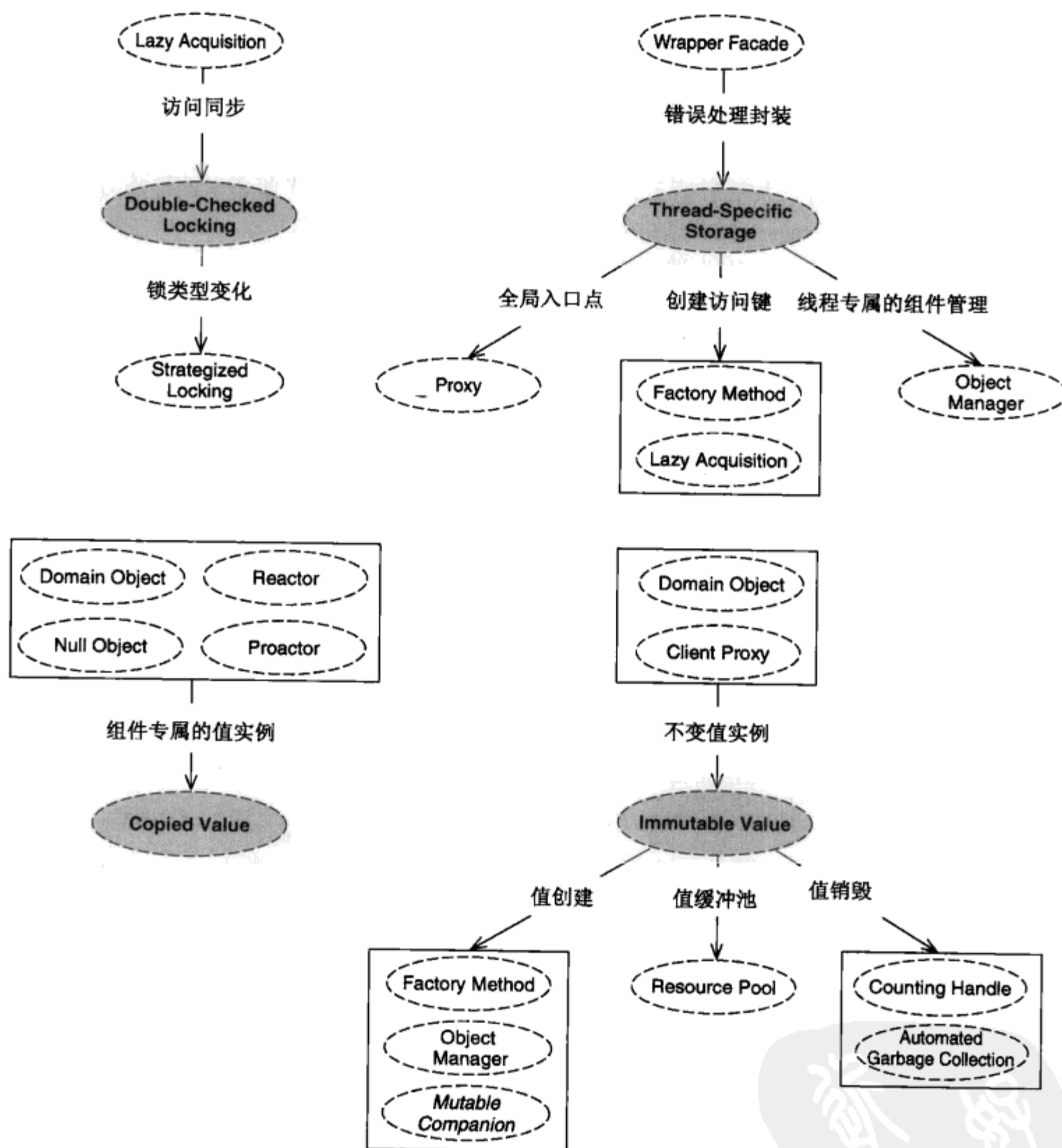


图 16-3

16.1 Guarded Suspension**

在实现Monitor Object (214)、Future (223) 或者被多个线程共享的其他组件的时候……我们必须能够合理地调度多个线程的执行顺序使其相互协作。



在并发性程序中，我们经常只在特定条件——称为守护条件（Guard Condition）——满足时才执行组件上的方法调用。组件上的某个方法不能立即执行并不代表我们就可以将其中止，这是因为其守护条件的状态可能因为同一组件上的另一并发行为而变成真的。

例如，我们只能在队列不为空时才能从同步的消息队列中移除一条消息，这就是守护条件。如果另一个客户端线程将一条消息插入到队列，那它就不再为空了。先前调用的移除方法因为守护条件是“队列空”而不能执行，现在就可以继续执行并成功完成了。

无限期阻塞是行不通的，因为那会妨碍其他客户端线程在共享组件上执行有用的工作。无条件的阻塞还可能造成死锁，因为所有的客户端都被锁定了，包括那些可以将守护条件改为真的操作。

中止对临界区的访问企图具有快速失败的好处，可以通过状态返回值或异常将失败信息发送给调用线程。这些信息对于客户端线程来说是有价值的，然而，客户端代码必须实现某种重试策略，这会使客户端段代码和实现细节混杂在一起，而事实上，这本应当被封装在别处。

因此，不要立即中止方法，而是将客户端线程挂起，以便其他客户端线程可以安全地访问共享组件并改变方法的守护条件状态。一旦状态发生变化，恢复被挂起的线程使其可以尝试继续执行中断的方法（见图16-4）。

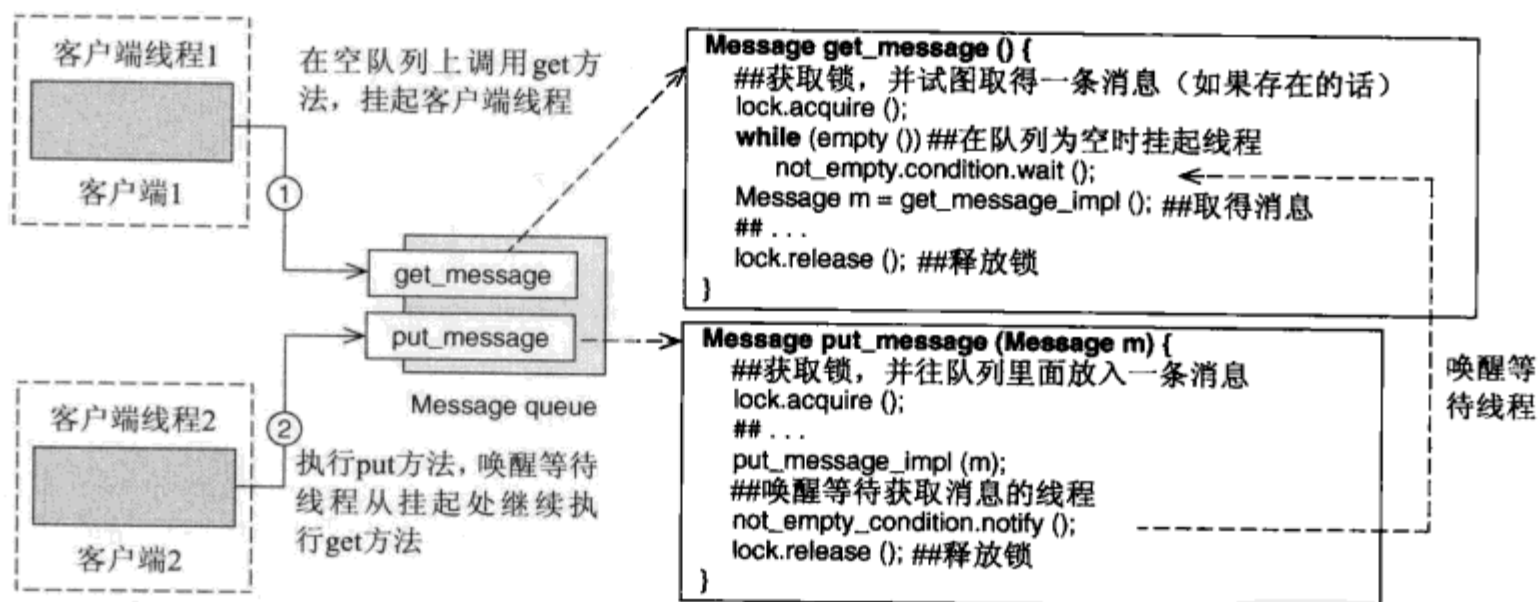


图 16-4

Guarded Suspension允许共享组件控制并调度客户端线程的执行顺序，这些对客户端线程是透明的。



Guarded Suspension设计的关键好处在于它对客户端是非侵入的。被调用方法封装了访问的策略和机制，从而避免扰乱调用代码或者在调用端出现重复。非异常的常规状态不再被标记为失败。即使在阻塞的情况下，共享组件仍然是可用的。如果挂起实现在操作系统层次，甚至都不会浪费CPU资源。这样的结果是Guarded Suspension降低了客户端线程的并发开销，并提高了共享组件的可靠性。

许多操作系统API都提供了Guarded Suspension实现原语 (primitive), 但是我们不应当直接使用这些原语, 而应当使用Wrapper Facade (269) 来封装。实现Guarded Suspension [Lea99] 有多种方式: 通过条件变量和相关的互斥件实现“等待并通知” (wait and notification)、通过“空转” (spin-loops) 实现“忙等待” (busy-wait) 或者直接将客户端线程“挂起/恢复” (suspending and resuming)。

尽管非侵入和非失败阻塞 (non-failing blocking) 可以简化客户端代码, 客户端可能希望在阻塞和做些其他什么之间做出选择, 特别是在挂起时间很长时。这一点可以通过为方法提供一个非阻塞的超时变量来解决。例如, 一个队列可以提供一个阻塞式的get方法和一个非阻塞式的try_get方法。

16.2 Future**

在实现Master-Slave (186) 结构、Active Object (212)、Partial Acquisition (303) 的时候, 或者软件中的客户端和服务端并行运行并通过方法调用来通信的时候……我们经常需要访问一些特殊的数据, 其计算过程与客户端的控制流是并发的。



16

组件上并发调用的服务可能需要向客户端返回结果。然而, 如果客户端没有在调用服务后阻塞, 而是继续进行自己的运算, 那么当客户端需要使用服务结果时服务器端可能还没有得出结果。

并发性的一个常见用途是通过交叠运算和通信来优化性能。一种简单的优化方式是使用不需要返回结果的单向调用, 即“即调即忘 (fire and forget)”的方式。然而, 客户端可能需要在一个月或多个服务器上调用一个或多个双向方法而不用同步等待服务器的响应。简单的“调用-返回”的处理方式, 在这种情况下无法使用。然而, 当客户端需要结果以继续处理时, 必须有一种直观的方式来获取结果。

因此, 当调用服务时, 立即返回一个“虚拟的”数据对象——称为Future——给客户端。该Future对象记录服务并发计算的状态, 并在计算完成后向客户端提供具体值 (见图16-5)。

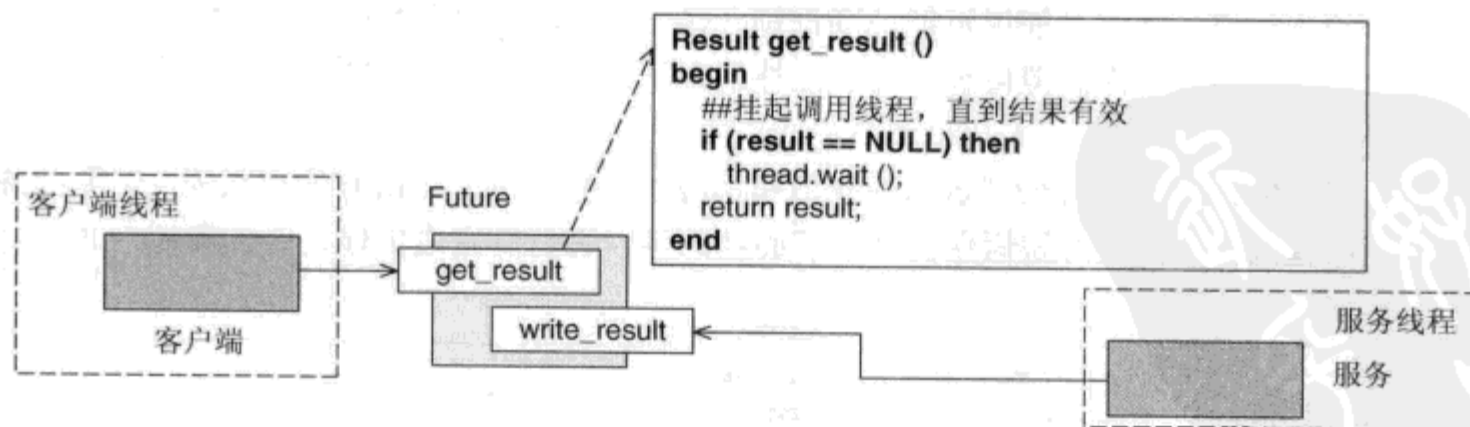


图 16-5

如果客户端在服务结果可用前对其进行访问, 则Future对象挂起客户端线程。当结果可用并存储在该对象中后, 客户端线程将自动恢复, 继续执行并使用服务结果。客户端也可以通过非阻塞的或定时的访问器以及完成状态查询来检查Future对象的状态, 这就使得我们可以在客户端检

查结果是否可用，而不用无限期地阻塞客户端线程。



使用Future模式可以增强系统的并发性，因为客户端在最后时刻才需要跟自己调用的服务进行同步。使用Future模式时，我们处理访问结果的顺序可以和方法调用的顺序有所不同，这可以提高系统的灵活性和性能。

为了提高并行能力，不要在得到Future对象后立即访问它，因为这种双向服务调用的同步方式代价非常高。我们应该让客户端在调用服务后执行尽可能多的操作和指令，直到客户端没有Future对象就不能执行任何操作时才访问它。在服务调用和访问Future对象之间间隔的时间越长，客户端被阻塞的可能性就越小，从而可以增加系统的并发和并行能力。

对于尚未完成计算的服务结果数据而言，Future对象相当于一个Virtual Proxy (294)。它可以直接用支持Guarded Suspension (221) 的锁定原语来实现；也可以建立在用Asynchronous Completion Token (155) 表现的事件处理机制之上。

16.3 Thread-Safe Interface*

在Requestor (140)、Client Request Handler (143)、Message Channel (130)、Message Router (134)、Shared Repository (117)、Explicit Interface (163)、Proxy (169)、Page Controller (196)、Front Controller (197)、Application Controller (198)、Monitor Object (214)、Object Manager (291) 和Builder (312) 模式中……我们往往需要保证在并发性程序中对组件的访问是线程安全的。



并发性程序中的组件必须是线程安全的。通常它们的方法以获取锁的方式保护对临界区的并发访问。然而，如果某个方法获取的锁是非递归的，并且该方法所调用的组件中的另一个方法也试图获取同一个锁，这时就会出现“自死锁”。

尽管可重入锁 (re-entrant lock) 可以防止自死锁，但对某些平台来说在组件内部方法调用中频繁地获取和释放锁会引入不必要的开销。理想情况下，我们的设计既要避免自死锁，而且不需要使用支持重入的锁，同时又能尽可能地降低锁开销。然而，我们不希望将共享组件的同步交给其客户端来完成，因为这样会使得客户端和组件耦合紧密。这样的耦合增加了使用的复杂性以及出错的可能性。

因此，将组件方法分为可公开访问的接口和相应的私有实现。接口方法获取锁，调用相关实现方法，然后释放锁。实现方法假定已获取了必要的锁，它负责完成工作，而且实现方法只调用其他实现方法（见图16-6）。

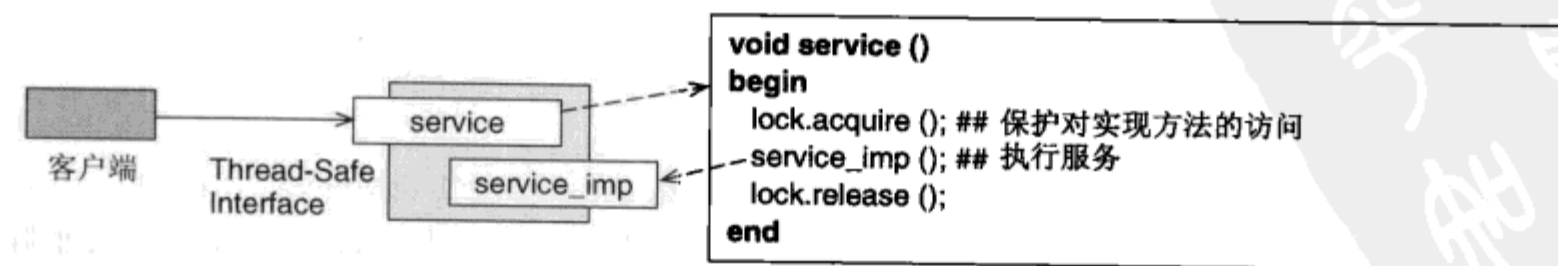


图 16-6

为了保证能够正确同步，采用Thread-Safe Interface设计的组件客户端只能调用其接口方法。一旦接口方法获取到必要的锁，它便将控制转交给相应的实现方法，再由实现方法处理客户端请求。这样就避免了自死锁和加锁开销，因为实现方法并不获取组件锁，它所调用的组件内的其他实现方法也不会获取组件锁。当控制流从实现方法返回到接口方法时，接口方法释放其获取的锁并将结果返回给调用该方法的客户端。



Thread-Safe Interface确保不会因为组件内方法调用而产生自死锁。此外，也不会出现无谓地获取和释放锁的情况。最后，Thread-Safe Interface将加锁和功能问题分离开，有助于各自的简化。

但是请注意，如果控制流会暂时离开组件范围，并且底层存在非常严格的锁定机制时，Thread-Safe Interface并不能完全解决自死锁问题。如果某个组件的实现方法将控制流转交给另一个组件，而那个组件又试图调用第一个组件的接口方法重新进入该组件，这时还是可能会发生自死锁。在这种情况下，接口方法会试图再次获取组件已获得的锁。解决这一问题的常见方法是使用可重入锁实现Thread-Safe Interface。

通常具有Thread-Safe Interface的组件会被设计成带有公开同步接口方法和私有非同步实现方法的形式。另一种做法是将Thread-Safe Interface实现为Template Method (265)，其中模板方法相当于接口方法，而钩子方法相当于实现方法。组件可以使用Stratigized Locking (226) 来改变其配置，以使用最恰当的锁类型。

16.4 Double-Checked Locking

当我们在并发性环境中实现Lazy Acquisition (300) 的时候……可能经常需要在某个方法中完成线程安全的、一次性初始化工作，而不必在方法的所有后续调用中进行同步。



如果组件由多个线程共享，要避免组件内出现竞争状态，常见的方法是将对其临界区的访问进行串行化。希望进入临界区的线程必须首先获取锁。然而，如果对象的临界区只会在某些条件下执行一次，但却会被频繁进入，这种设计会引入过多的加锁开销。

例如，某个对象可能是由Lazy Acquisition模式进行初始化的。我们首先检查对象是否存在，如果不存在则创建之。在多线程环境中，这种即时初始化代码属于临界区，它在对象的生命周期内只会执行一次。但是将这段代码用锁保护起来会为所有访问方法引入开销，而不只是初始化实际发生的那一次。因为对对象是否存在的检查本身就可以防止控制流执行临界区初始化代码，所以锁获取和释放是没有必要的。

因此，为共享组件提供一个“标示”以反映特定临界区是否有必要执行。在获取守护临界区的锁之前和之后检查该标示（见图16-7）。

如果临界区代码已经执行过了，对标示的首次检查会跳过这段代码和相关的获取锁操作，因而，不会产生加锁开销。对标示的第二次检查防止并行的两个或多个通过初次检查的线程产生竞争状态。只有其中的一个线程能成功获取锁并通过对标示的第二次检查，执行临界区并改变标示。一旦锁被释放，任何其他等待线程都将跳过临界区，因为第二次检查表明它已经执行过了。

```

## 第一次检查“标示”的值
if (first_time_in_flag is FALSE)
    获取锁
    ## 再次确认“标示”的值，以避免竞争状态
    if (first_time_in_flag is FALSE)
        执行临界区
        set first_time_in_flag to TRUE
    fi
    释放锁
fi

```

图 16-7



在对象内部，Double-Checked Locking确保只在条件语句保护的临界区必须执行的情况下获取和释放锁，这通常是“例外的”情形。而不执行临界区的“常规”情形不需要加锁，因而执行得更快速高效。

Double-Checked Locking所使用的标示的初始值应当表明临界区还没有被执行。如果该标示还有应用相关的目的，例如是指向内部表现对象（an internal representation object）的指针，那么就要确保它是原子类型的。通过检查标示来保护临界区，只允许线程在要执行临界区时进入。在临界区内获取锁，再次检查标示，并根据它决定是否执行临界区。在释放锁之前，改变标示的值，从而保证后续线程不会再次进入临界区。Strategized Locking (226) 支持为应用配置适当类型的锁。

如果没有平台能够支持一种内存模型来保证内存更新的视图具有一致性，我们就需要CPU相关的指令，如内存屏障（memory barrier）来安全地访问该标示。目前没有可移植的C++内存模型符合我们的要求，因此必须采用平台相关的方案。当前版本的Java支持一个合适的内存模型，但是老版本没有。正如其他免锁技术一样，Double-Checked Locking中的双重检查可能会非常精巧并且容易出错。开发人员需要很小心其中的微妙[MeAl04a][MeAl04b]。如果不可能访问这种层次的机制，或者不现实，则应当考虑其他设计，特别是那些能完全避免加锁的设计。

16.5 Strategized Locking**

在Requestor (140)、Client Request Handler (143)、Shared Repository (117)、Page Controller (196)、Front Controller (197)、Application Controller (198)、Command Processor (199)、Thread-Safe Interface 和Double-Checked Locking模式中……多线程编程的关键问题之一是为特定环境选择合适的锁定策略。



在多线程环境中，线程共享的组件必须保护其临界区不会被并发访问。然而，不同的软件配置可能需要不同的锁定策略，比如互斥体、读-写锁或者信号量。

要解决这个问题，我们可以将锁定策略直接在组件中做硬编码，并根据特定环境进行调整，但是这对大多数应用来说并不可行。组件将会依赖于环境，而且一旦环境发生改变需要不同的锁定策略时，所有组件都需要更新，这会引入相应的维护成本。为每个环境提供不同的组件版本也不可行——它带来类似的维护开销。理想情况下，应当可以定制组件的锁定策略，而组件的实现

不依赖于具体的环境。

因此，采用“可插拔”方式定义锁，每种类型的锁代表特定的同步策略。为所有类型提供一个公共接口，这样组件可以统一使用所有锁类型，而不会依赖于其具体实现（见图16-8）。

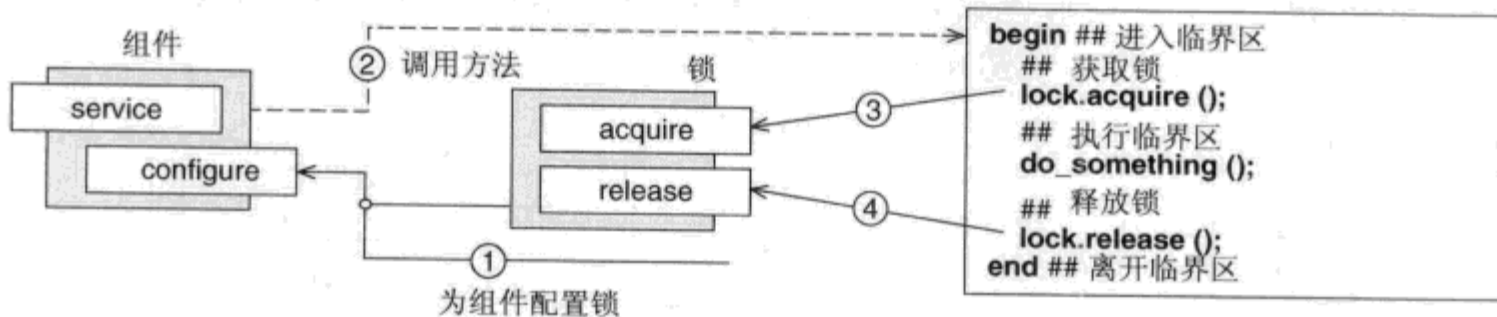


图 16-8

在创建或声明时，为组件配置适当类型的锁实例，例如在构造函数中传入锁对象或者用特定锁类型作为组件类的参数。使用该锁实例来保护组件中的所有临界区。



Strategized Locking设计有很多优点。与单独为每一种并发模型提供实现不同，它只有一种核心实现。因此，对组件的改进和错误修正不需要重复。为组件配置和定制特定的并发性模型非常简单，并且对组件是非侵入的，因为组件的同步方面是可以策略化配置的。相反，Strategized Locking向组件使用者暴露出了一个可参数化的设定，通常可以认为这一点是具有侵入性的^①。然而，正是这种开放而且正交的方案使得组件可以在其他的应用环境中使用。

为了使得锁“可插拔”，定义一个锁获取和释放的Strategy (266) 接口，该接口由所有具体锁类型实现。在同一应用中使用相同的锁类型配置相关组件。当不同的锁实现与平台相关而不是与锁策略相关时，采用Wrapper Facade (269) 实现具体锁类型，这样可以封装与特定平台相关的加锁机制细节。为了在不需要加锁的单线程环境中优化组件，提供一个空（Null）锁定类型，即获取和释放锁都是空操作的Null Object (267)。

Scoped Locking (227) 有助于在组件实现中简化并自动完成安全的获取和释放锁的操作。

16.6 Scoped Locking**

在并发性程序中，为了给共享组件提供加锁机制，无论采用何种方式——是将特定锁类型硬编码到组件中，还是实现为Strategized Locking (226) ……多线程编程的关键之一是确保在进入和离开临界区时自动获取和释放锁。



临界区的代码序列通常需要由锁来保护，当控制进入和离开临界区时必须获取和释放锁。但是，如果程序员必须显式获取和释放锁，那么将很难确保代码的所有路径都释放了锁。

控制可能因为遇到未捕获的异常或者显式的“退出语句”，如return、break或者goto等，而

① 这两句话分别指出了Strategized Locking对组件的“非侵入性”和对组件使用者的“侵入性”，注意区分。——译者注

过早地退出。程序员可能因疏忽而遗漏掉释放锁的代码。代码越复杂繁琐，这种疏忽就越有可能发生。试图通过显式使用类似于try、catch和重新throw等语句实现异常安全的代码很容易出现问题，进而失去简洁性和所期待的安全性。

因此，划定临界区的范围，在控制进入该范围时自动获取锁。同样，在控制通过任意退出路径离开该范围时自动释放锁（见图16-9）。

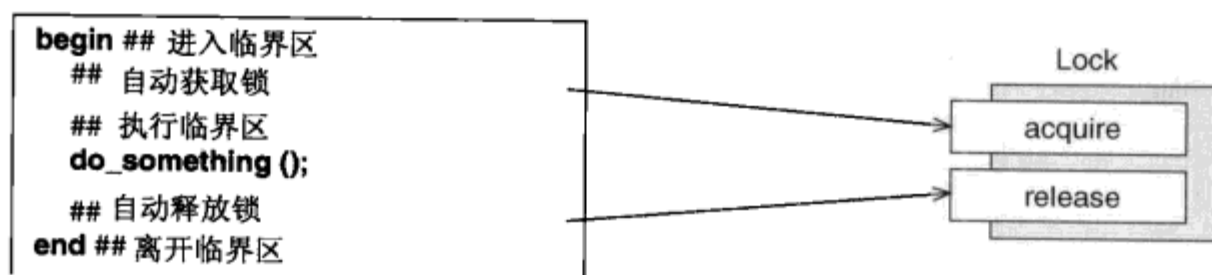


图 16-9

这样，进入临界区将会是线程安全的，而离开临界区时也会安全地释放所有已获取的锁。



Scoped Locking消除了与同步及多线程相关的常见编程错误，增强了并发性软件的健壮性。而且，临界区由编程语言的方法（method）和代码块定义，控制进入和离开临界区时可以自动地获取和释放锁。

Scoped Locking的实现依赖于编写并发性软件所使用的语言。例如，Java提供了专门的语言关键字——synchronized，用它来指示Java编译器产生相应的字节码指令块，并用monitorenter和monitorexit包围该指令块。为了确保能够释放锁，编译器生成一个异常处理程序，用来捕获该同步块中抛出的所有异常[Eng99]。如果锁定范围包含整个方法，方法本身可以标记为synchronized。因为标记方法比写一个同步块要简单，我们很容易倾向于依赖更简单的方法。不幸的是，在很多情形下，这意味着锁定范围比临界区要大，从而导致并发性的损失。

和Java不同，C++并没有对Scoped Locking提供直接的语言支持，但它可以通过Execute Around Object (264) 提供C++方式的实现。这种方式创建一个守护类，其构造函数获取锁而析构函数释放锁。因此，将守护对象声明为临界区范围内的一个本地变量，并放到临界区第一个语句之前。这样当控制进入临界区的时候，守护类的构造函数被调用并获取锁；当控制通过任意退出路径离开临界区时，根据C++的语义，守护类的析构函数自动被调用，从而释放锁。将守护对象设计为Wrapper Facade (269) 有助于用统一的接口封装平台相关的具体锁定机制。

16.7 Thread-Specific Storage

如果我们所使用的代码在开发的时候并没有考虑到多线程或者并不是为某种集成环境所设计，或者在使用Wrapper Facade封装UNIX环境中运行的C/C++遗留并发性系统的错误处理机制时……我们通常认为使用的状态对应用是全局性的，但是，在并发性实现中，它应当表现为在物理上是位于每个线程中的。



有时候我们访问的对象是和环境绑定在一起的，这使得它看起来在逻辑上是全局的。然而，如果我们需要环境看上去是针对每个线程的，那么对象不能简单地在物理上是全局的，即不能只有一份状态副本。

为对象的每次访问加锁，并根据线程查找合适的值，这在对象使用频繁时会降低系统性能。理想情况下，访问对象应当是原子的，不需要引入任何加锁开销。此外，更新其实现以支持多线程通常也不可行，许多遗留组件编写时根本没有考虑多线程，而是依赖于特定对象在一定程度上的全局性。

因此，为环境绑定对象引入公共入口点，而将其物理对象实例保留在线程本地的存储空间中（见图16-10）。

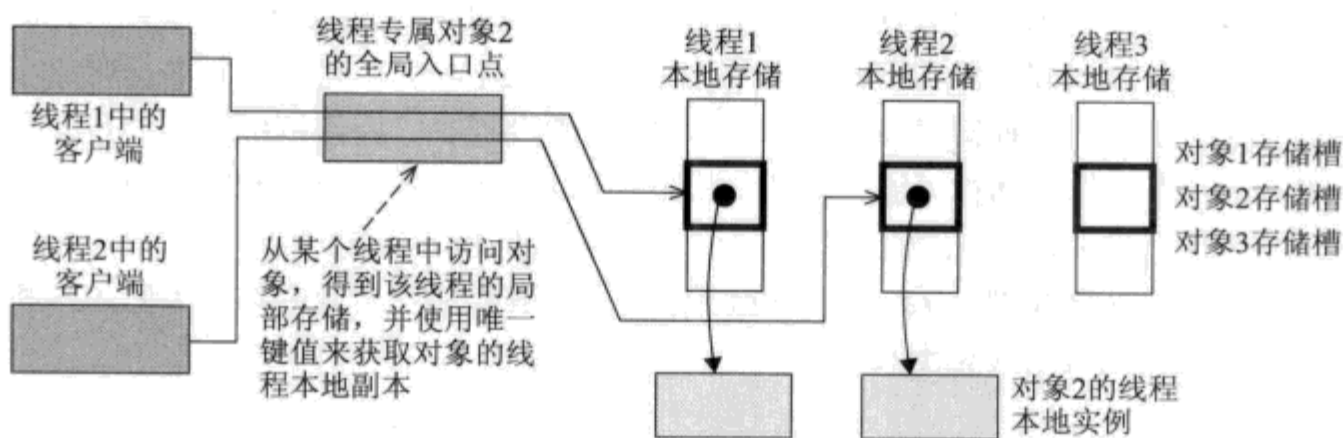


图 16-10

确保对象的每个线程本地副本可以通过全局唯一的键值从相应的线程本地存储中获取。使用该键值，将对象全局入口点上的所有方法调用转发给线程相关对象的特定副本，该副本由调用入口点的线程负责维护。因此在访问线程本地对象的副本时不需要加锁。让客户端线程只能通过全局入口点操作线程相关的对象，以保持对象在逻辑上的全局可见性。



Thread-Specific Storage的设计在访问线程相关数据时不需要加锁。此外，软件开发人员可以很方便地使用Thread-Specific Storage来处理遗留代码中关于对象全局性的假设，也可以用来处理对分布式的上下文信息未提供合适参数的情形。然而，该模式并非没有缺点，其中最显著的问题是它鼓励开发人员将那些本应当考虑为“本地的”并作为参数来显式传递的对象设为“全局的”。

将线程相关对象的全局入口点实现为Proxy (169)。在运行时，在每个线程中将线程相关对象的全局入口点做实例化，而线程本地实例则推迟做实例化。当客户端线程调用入口点的方法时，Factory Method (313) 使用Lazy Acquisition (300) 检查是否存在全局唯一的键值来标示该线程相关对象，如果没有则创建一个。最后，将入口点上的调用转发给该线程相关对象并执行请求。注意，不需要使用锁来保护对对象的存在性检查、创建及其使用的并发性访问。尽管客户端通过逻辑上的全局入口点来激活调用，实际上它们是在线程间非共享对象状态上操作的。

多个线程相关组件可以维护在一个索引容器中，比如一个Map或者Object Manager (291)，其索引类型是键值类型，并为每个线程提供一个单独的实例。

Copied Value就需要手动地在调用时显式创建副本，因此容易出错。

16.9 Immutable Value**

在通过Domain Object (121) 来表达细粒度值的时候，或者设计Client Proxy (139) 的时候……我们希望能在应用的不同线程间高效而安全地传递值。



值对象的引用通常是分布并储存在多个字段（field）中的。然而，如果多个对象共享同一个值实例，其中一个对象导致值的状态发生变化，可能会对其他对象产生意外的影响。对值进行复制可以降低同步的开销，但是会引入对象创建的开销。

对于可改变的值对象来说，复制可以减少别名带来的问题。但是如果没有适当的语言支持，这种做法是繁琐且容易出错的。这还可能导致创建过多的小对象，特别是在值被频繁查询或者在组件间传递的情况下。理论上，这些值没有发生变化，所以创建值的多个实例只是为了通信，根据底层对象创建模型的不同，可能会比较浪费并且会引入过多的空间和时间开销。

在多线程环境中共享和状态改变的问题会成倍的增加。方法同步解决了每个独立修改的有效性问题，但是对于有关共享的一般性问题则无能为力。对改变进行同步还会引入性能开销。

因此，定义一个值对象，使其实例是不可改变的。值对象的内部状态是在创建时设置的，并且不允许后续改变（见图16-12）。

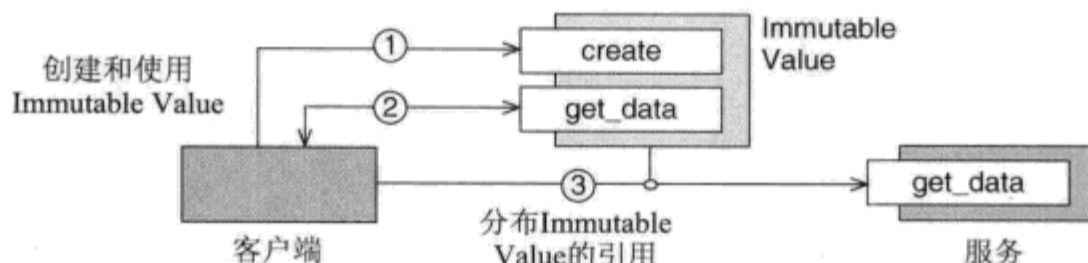


图 16-12

在Immutable Value对象中只提供查询方法和构造函数，不会定义修改函数（modifier method）。因此，值的改变就成了切换所引用的值对象。



没有任何状态改变意味着不需要同步。这本身就决定了Immutable Value是线程安全的，而且因为没有加锁机制，也意味着它在多线程环境中的使用是高效的。在其他环境中共享Immutable Values也是安全和透明的，因此不需要复制Immutable Value。

在Java中，将Immutable Value的字段定义为final以确保它不会被改变。这还意味着要么该类本身必须是final的，要么其子类必须也是Immutable Values。在C++中，一个全部用const限制的（fully const-qualified）对象可以扮演相似的角色，可以通过指针来分发和共享。然而，对进一步继承的限制，必须通过惯用法来表达，而不是语言机制。

如果需要一个具有不同属性的值，则创建一个新的对象或者找到一个符合期望的对象——我们只能改变值的引用，而不能改变其属性。有一些辅助技术来创建Immutable Values。可以通过

一个完整直接的构造函数，或者通过几个类级别的Factory Method (313) 来创建。Mutable Companion[HenOOB]可以操作其他值对象，并为其结果创建一个Immutable Values，例如计算给定值的两倍。Resource Pool (298) 有助于提供对一套预先定义的Immutable Value的访问，而Object Manager (291) 支持创建单实例的Immutable Value。

值对象并不代表资源，所以不像资源对象那样，存在对象销毁的问题。我们可以使用Automated Garbage Collection (307) 回收不再使用的Immutable Value的资源。相反，在没有垃圾回收机制的环境中共享Immutable Value需要小心管理——线程生命周期必须小于所共享的Immutable Value的生命周期，否则就需要通过Counting Handle (309) 来引用。





Kevlin和Frank在JAOO 2006大会上，就对象进行交流
©Mai Skou Nielsen

涉及应用架构的问题并不都是像调用另一个组件内对象的方法这样的简单。很多情况下，应用定义了一个可供他人使用的框架，可以是作为扩充，也可以以插件的形式存在。封装了交互模型的框架比起仅提供简单、被动对象类型的组件，更有可能带有较复杂的对象之间的处理逻辑设计。本章展示了一系列支持对象间交互的模式，这些对象可以位于应用、框架或产品线的不同组件中。

构建在组件框架上的应用，可以利用框架的执行、资源及其关系管理的特性，同时应用需要遵循框架的协作协议。在这样的应用中，对象之间的协作往往不是简单地在对象之间进行同步方法和服务调用，还要随着函数调用传递参数以及通过返回值收集结果。虽然组件对象之间的协作常常遵循这个简单模型，但即使在顺序编程中，该模型也不会从头到尾一直保持。

在设计这种交互时，下面一些问题经常会出现，有些非常普遍，有些则仅限于分布式环境中。

- 解耦。在框架、产品线和所有大型长生命周期的系统中，组件之间常常是松耦合的，目的是防止不必要的依赖关系，并对它们的独立改进和重用以及与高层服务的组合提供支持。组件之间的松耦合也为它们之间的交互提供了帮助：尽量降低对特定互操作协议和

策略以及用于组件之间交互的数据结构的显式依赖。组件提供的行为依赖于其调用者的类型时，同样会对解耦有所要求。通过组件代码内部的条件语句来解决这种问题是可能的，但要避免结构的复杂并更好地支持代码的维护和演化，我们不应该对这样的依赖关系进行硬编码。

- 协调一致性。在软件系统中，组件之间可以脱离彼此独立运作，特别是在分布式和并行软件系统中更是如此。但是，有时候会需要一些组件以一致的方式来对其他组件进行协调，例如，避免其他组件内部状态的不一致，或是根据特定的高层协作或集成场景来安排组件的执行顺序。
- 通信开销。较之独立系统，分布式系统中组件对象之间的通信可能导致更高的延迟和阻塞。例如，当前网络负载越高，客户端与远程组件对象之间的消息交互就需要花费更多的时间。因此，设计高效、可扩充的分布式系统的一个关键目标就是尽量减少并优化网络通信。

因为上面这些问题，无论是在开发分布式或并行软件时，还是在创建框架或定义产品线架构时，应用的设计者必须仔细考虑组件对象彼此之间如何协作。在我们的分布式计算模式语言中，有8种模式可以通过引入高效而又灵活、紧凑的对象互操作，来帮助解决这些问题。

- Observer（观察者）模式 (237) [GoF95] 通过支持状态变更的单向传播来辅助互操作对象之间的状态同步。当一个组件对象的状态改变时，其Observer会得到通知。
- Double Dispatch（双分派）模式 (238) [Beck97] 可以帮助组织组件对象之间的通信，在该模式中，被调用对象的行为取决于调用者对象的类型。
- Mediator（中介者）模式 (239) [GoF95] 封装了组件对象集合互动的方式。Mediator通过避免对象显式地相互引用使其松散耦合，而且可以独立地改变它们之间的交互。
- Command（命令）模式 (240) [GoF95] 将一个请求封装为对象，这使得我们可以把不同的请求作为客户端的参数，并支持可取消操作。
- Memento（备忘录）(242) [GoF95] 支持在不破坏封装性的前提下，捕获一个组件对象的内部状态，并在该对象之外保存这个状态。
- Context Object（上下文对象）模式 (243) [ACM01][Kel04][KSS05][Hen05] 以组件对象的形式捕获环境服务和信息，以支持将这些信息传递给其他需要获取执行上下文的服务和插件对象。
- Data Transfer Object（数据传输对象）模式 (244) [ACM01] [Fow03a] 通过将一组属性打包到简单对象中，并在单次调用中传入和返回，以减少远程组件对象的更新或查询调用的数量。
- Message（消息）模式 (245) [HoWo03] 将两个应用组件对象交换的信息封装到一个数据结构中，而该数据结构可以通过网络传播。

本章中这8种对象交互模式可以划分为两类：协作和数据交互。

其中的协作型模式包括Observer、Double Dispatch、Mediator和Command，它们可以帮助协调和安排应用中组件和对象之间的互操作。如上面的概览所示，其中每一种模式都针对该上下文的一个特定方面。因此，所有4种模式彼此之间互为补充，而不是替代的关系。

图17-1展示了如何将协作型模式集成到我们的模式语言中。

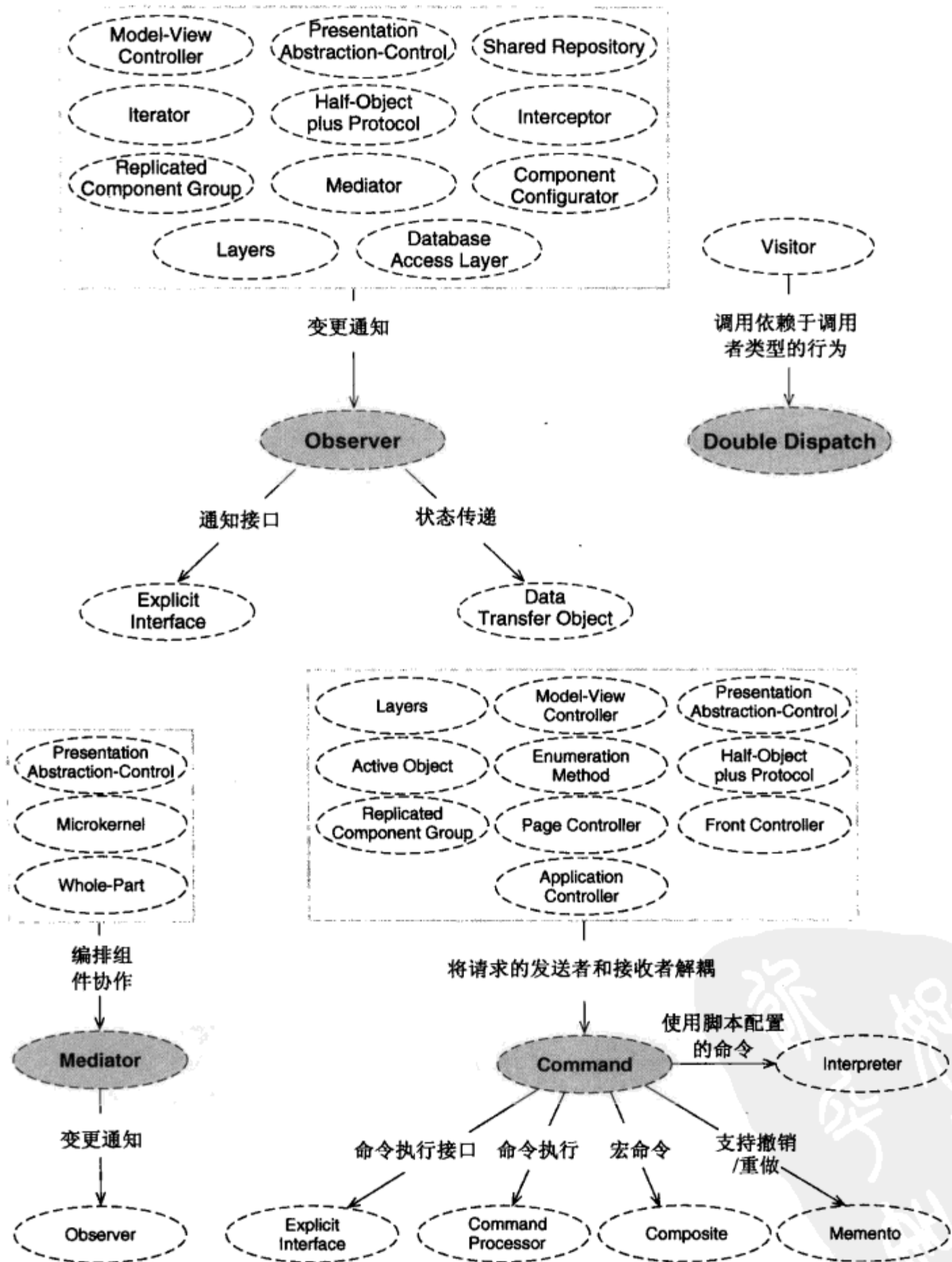


图 17-1

4种数据交互模式包括Memento、Context Object、Data Transfer Object以及Message，它们同样瞄准了对象交互的不同方面。因此，在设计组件接口时，它们常常结合起来使用。例如，客户端在调用方法时传递一个Context Object到组件中，并接收一个封装了调用返回结果的Data Transfer Object作为返回，或者，当返回结果是组件内部状态的快照时，接收一个Memento对象作为返回。Message可以表示其他3种模式的实例，同时它也可以用来表示一个服务请求或序列化的Command，这使得它们可以使用特定的协议在网络上进行传输。

Memento可以概括Client Session State、Server Session State及Database Session State [fow03a] 3种模式，它们分别定义了客户端会话状态存储的不同位置，并使用独立的对象进行封装。Context Object模式在多种资料中均有描述，它还能用于处理很多本书中未涵盖的模式变体 [ACM01][Kel04][KSS05][Hen05]。

这4种数据交互模式联系到我们的模式语言中可描绘成图17-2。

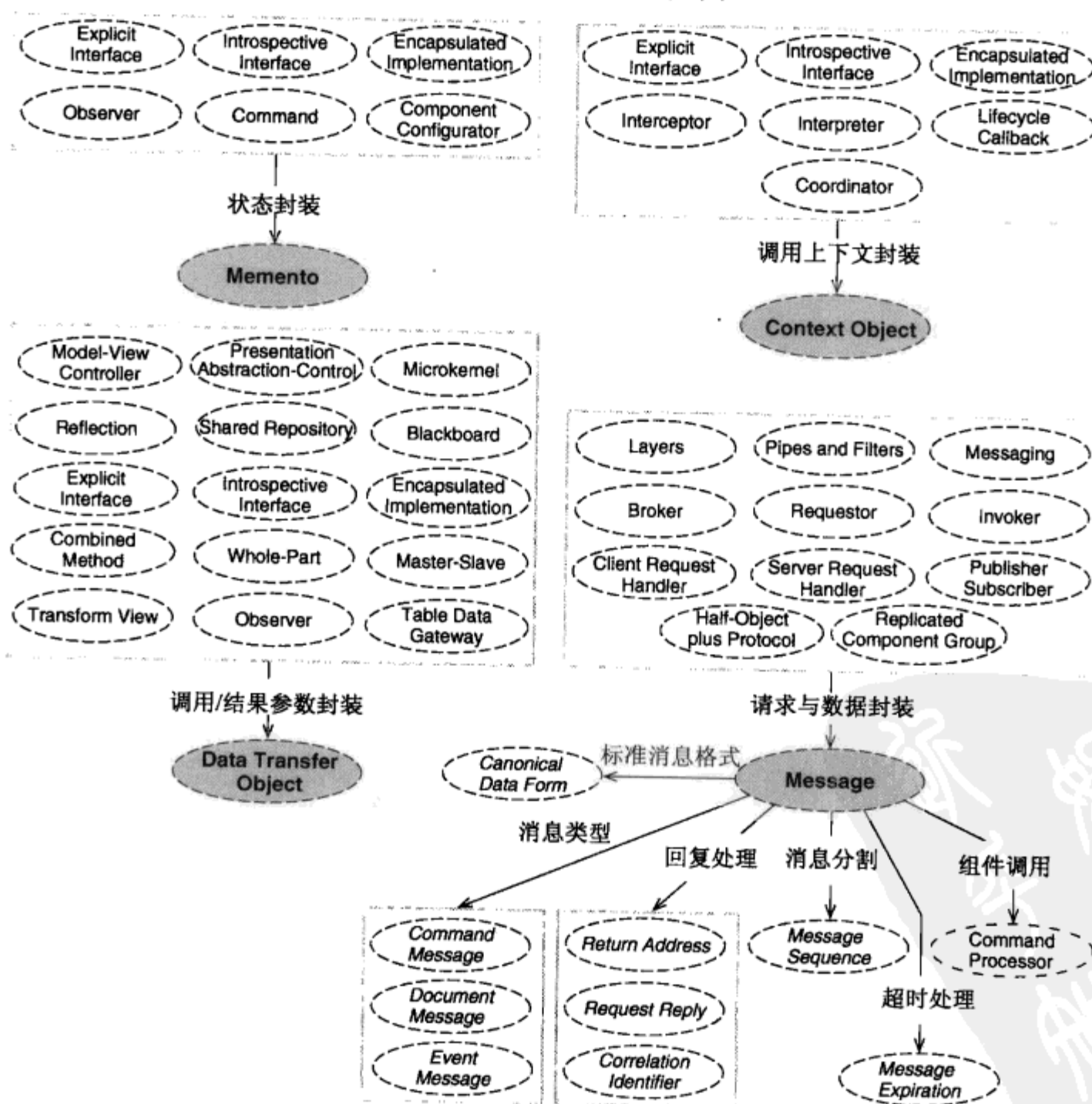


图 17-2

17.1 Observer**

在Layers (108)、Model-View-Controller (109)、Presentation-Abstraction- Control (111)、Shared Repository (117)、Iterator (173)、Half-Object Plus Protocol (188)、Replicated Component Group (189)、Iterceptor (260)、Mediator (239)、Component Configurator (289)以及database Access Layer (318)等模式的应用中……我们必须提供一种方式以使得协作组件对象集合中成员之间的状态保持一致。



有时，消费者对象（Consumer Object）依赖于其他提供者对象（Provider Object）的状态或其维护的数据。如果提供者对象的状态未经通知就发生了变更，依赖于它的消费者对象们的状态可能会变得不一致。

对这一问题的常见解决方案是由消息提供者对象发起，建立到所有依赖于它的消费者对象之间的硬性连接；或者使消费者对象从提供者中取得状态。但是，这些方法往往都不太实用，因为一个消费者对象可能并不直接依赖于提供者，在应用的生命周期中也可能出现各种不同的消费者。而且，采用消费者取信息的方法可能会消耗更多资源，也有可能无法尽快地取得状态的变更。

因此，定义一种变化发布机制，每当信息提供者——称为“Subject”——的状态发生变化时，即通知注册过的消费者——称为“Observer”，被通知的Observer可以对此作出必要的响应（见图17-3）。

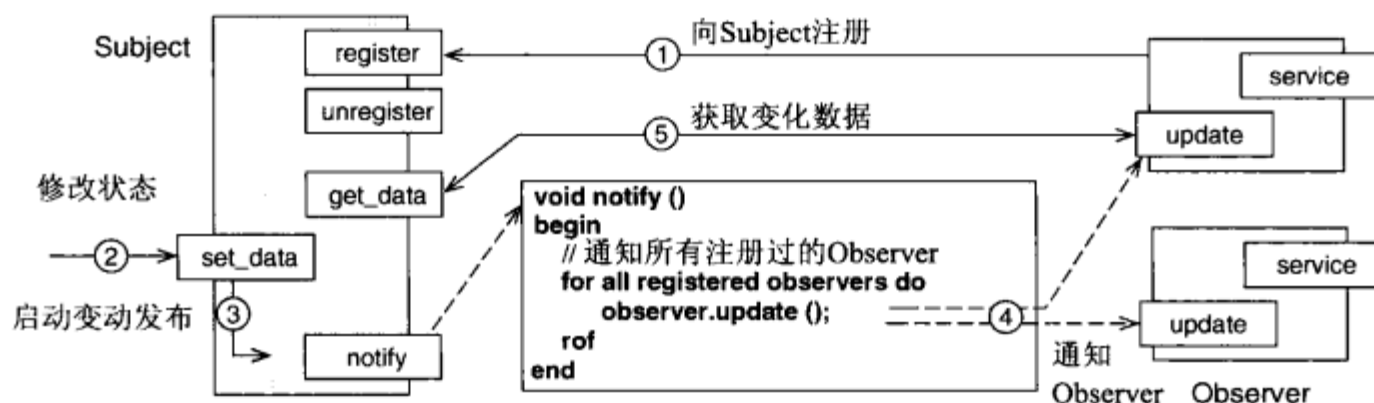


图 17-3

Observer必须定义一个特定的更新接口，通过该接口，Observer获得Subject状态变更的通知。该接口是Subject与其Observer之间的主要耦合。Observer可以动态地向Subject进行注册或注销。当Subject通知其Observer时，它可以将状态信息随着变更通知消息一起推给Observer，或者Observer可以在得到通知后，有选择地从Subject取回信息。



在Observer设计中，用Observer的动态注册来实现变更通知机制，以避免Subject与其Observer之间的硬编码联系，Observer可以随时加入或离开，这样新Observer可以集成到系统中，而无需改变Subject对象。Subject对象的主动变化发布避免了Observer主动查询Subject的变化，并确保Observer可以在Subject状态变更后立即更新自己的状态。

在典型Observer实现中，Explicit Interface (163) 定义了供Observer支持的更新接口。具体

Observer对象实现该接口以定义其特有的获得Subject通知时的更新策略。相应地，Subject提供接口以供Observer注册和注销通知机制。Subject对象在内部使用一个集合，如散列集（Hashed Set）或链表（Linked list），管理其注册过的Observer。

Observer模式的变更通知协议可以用多种方式来实现。最简单的选择是基于拉（Pull）模型：当Subject变更其状态时，它通知所有注册过的Observer状态发生了改变。得到通知的Observer可以随后回调Subject对象来取回更多细节信息。当所有Observer依赖于Subject维护的所有状态时，这种协议工作得很好。但当不同的对象依赖于Subject的不同状态时，该通用拉模型会导致不必要的更新，因为Subject会通知所有Observer，而不仅是那些依赖于变化状态的特定Observer。

如果Subject和Observer运行在不同的地址空间中，拉模型会引入不必要的网络和处理资源开销。在这种情况下，一种分类的拉模型可以用于允许Observer注册Subject的一种或几种不同类型的状态变更。这些Observer可以在特定类型的状态变更时才得到通知。由于需要让所有Observer回调Subject以取得状态，所以拉模型仍然不太适合于远程通信。

为了减少Subject与Observer之间的交互，我们有另外两种实现变更通知的协议，它们都基于推（Push）模型而不是拉模型。在常见的推模型中，Subject将其自身属性状态的快照随着通知推送给Observer，使用一个Data Transfer Object模式 (244) 来传递其属性。当所有Observer都依赖被推送的整个状态时，或者传递所有状态的开销小于每个Observer回调Subject特定状态的开销时，该模型是有用的。

通用推模型的一个变体是分类推模型，该模型使用了基于类型的过滤[GoF95]。如果Subject中的变更仅仅影响其整个状态的一小部分，则只向对之感兴趣的部分Observer进行推送。如果这些Observer对于某些变更通知并不作出任何响应，分类推模型也同样会产生不必要的开销。

在特定Observer配置中，选择最合适的变更通知机制同样受到耦合程度因素的影响。拉模型在Subject与其Observer之间造成的耦合通常会较推模型松，而通用推模型比分类推模型能更好地解耦Subject和Observer。

17.2 Double Dispatch **

在实现Visitor (261) 模式或要在具有继承关系的类之间通信时……我们必须实现依赖于两种类型的对象的行为。



很明显，一个方法的行为依赖其参数的值。但有时候，其行为也依赖于其参数的类型。大多数对象模型都不支持这种“多方法分派（Multi-method dispatch）”，而只支持单分派（Single-dispatch）。

多态常常用单分派来表示：方法的目标（Target）是调用的接收者，所调用方法的实现基于接收者的类型。但有时候行为既依赖于调用者的类型又依赖于被调用对象的类型，所以，调用者需要将自己作为参数传入。要表示这样一种配置，一种解决方案是根据参数的运行时类型进行显式选择，例如，在具体的接收者类中使用if……else或switch语句。但这种方式有些脆弱和冗余。另一种方式使用Map，其主键是由接收者和参数的运行时类型的组合，键值是某种形式的方法引

用，比如说C++中的成员函数指针，或是C#中的delegate。这种方式可能会造成不必要的脆弱性和运行时开销。

因此，将调用者对象作为附加参数传递给接收者对象。在接收者中，回调调用者对象以运行与调用者类型相关的逻辑，将接收者作为一个附加参数传递给调用者，以便调用者得以执行正确的行为（见图17-4）。

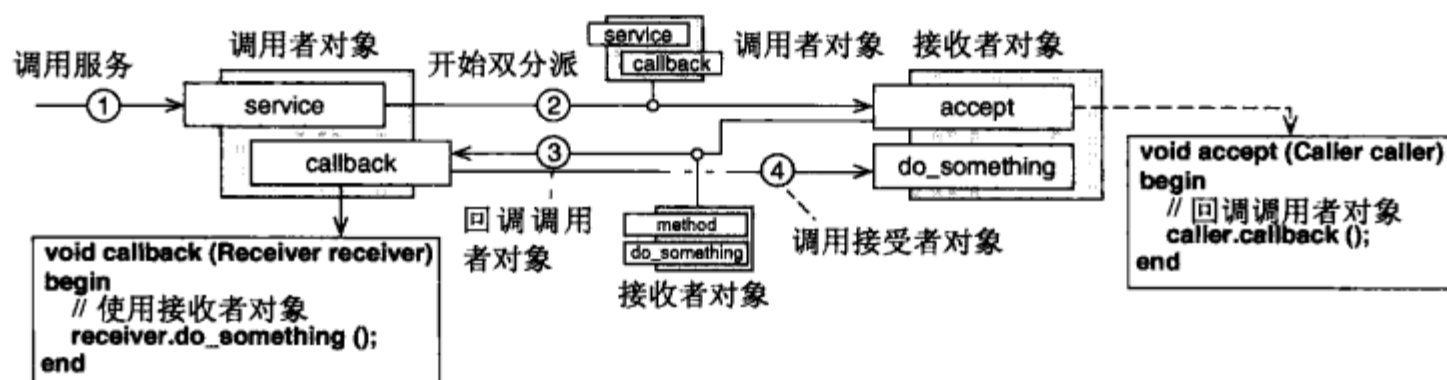


图 17-4

现在，行为分布在调用者和接收者对象中。调用者对象针对每个可能的接收者对象的类型有一个方法与之对应，可能是多个重载的方法，也可能是专门命名的方法。在实现调用者对象接口的时候，在每个接收者类型所对应的方法中定义了类型相关的行为。接收者对象负责选择与自己类型相对应的方法，并在调用时将自身作为参数传递给调用者。



Double Dispatch避免了在接收者对象中通过大量的if……else或switch语句来执行与调用者类型相关的行为。该方式难于维护和扩展，虽然看起来它是将所有的程序逻辑集中到了一处[Beck97]。在Double Dispatch模式中既涉及了调用者又涉及了接收者对象。这种回调的方式使得我们的代码更简单也更紧凑。

Double Dispatch模式的缺点是，它引入了远程对象之间额外的通信开销。而且，Double Dispatch模式的可维护性依赖于接收者类型继承关系的稳定性。如果要在接收者类型继承体系中添加一个新的具体类型，该参数接口及其所有的具体实现必须更新^①。根据该接口分布的广泛程度，这种改动可能会非常冗长乏味甚至于无法完成，因为组织界限、已发布API的多种版本、二进制稳定性等，都限制了接口改进的可能性。

17.3 Mediator*

在实现 Presentation-Abstraction-Control (111)、Microkernel (113) 或 Whole-Part (183) 模式的时候……我们必须减少多个互操作对象之间的耦合。



有时候，对象集合成员之间的关系是比较复杂的，集合中任意对象可以同其他几个对象进行

① 添加针对新的具体类型的回调函数，并实现之。——译者注

交互。但是，要让每个对象都自己维护所有的这些协作关系，可能会导致对象之间的过度耦合。

在随后对协作类的修改中，这种耦合所导致的互相依赖往往难以理解、测试和维护，因为向现有协作集合中添加对象的工作非常琐碎且容易出错。如果抛开相关的特定类型，这种协作在别处也会用到，这种紧耦合还可能对重用造成妨碍。

因此，通过Mediator对象封装集合中所有对象的聚合协作行为，从而将这些对象解耦合。协作以中介的方式间接地完成，而不再是点对点的直接通信（见图17-5）。

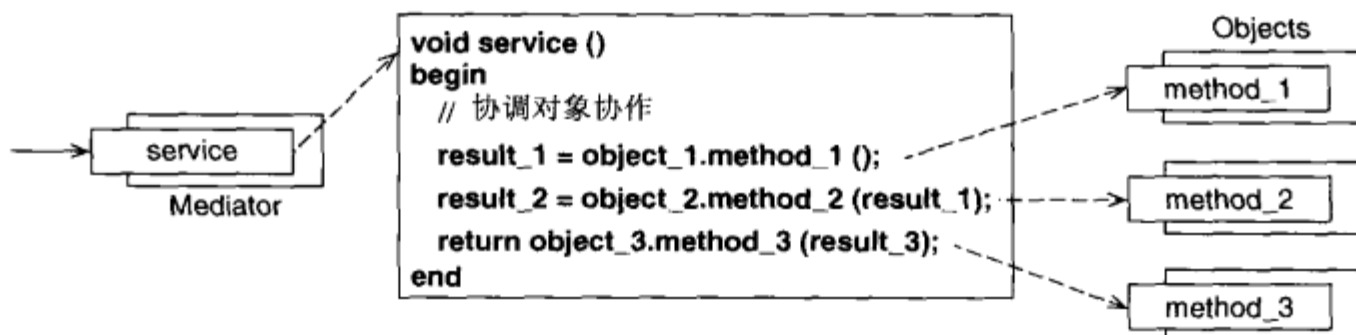


图 17-5

如果一个对象需要与另一对象进行协作，它可以通过向Mediator发送请求、消息或数据来实现，Mediator随后将信息匿名地路由到相应的接收者对象。相应地，处理结果可以通过Mediator返回给原请求对象。



Mediator对象使得多个协作对象保留其自我完备性和彼此之间的独立性，而不再需要对象进行显示的维护。相反，它们将请求路由、消息和需要与其他对象交换的数据委托给Mediator对象。在这里，Mediator作为一个编排者将协作对象联系起来，对其进行维护和监管，控制它们的协作。

虽然Mediator对象的引入保留了单个协作对象的内聚性、封装性和简单性，但控制的集中也使Mediator对象自身的维护成为一个潜在的问题。同样，Mediator对象也可能会成为潜在的性能瓶颈，进而造成分布式系统中的单点失败。

通常有两种方式来实现Mediator：一种是将对象之间的协作关系硬编码，另一种是可以将Mediator对象作为Observer (237) 模式中的Subject来为需要的对象提供请求、消息和数据。后者采用了一种隐式调用的风格，其耦合程度是很低的。但是，这种方式又使得实际的控制模块较难辨识，这反而比硬编码的设计更难实现。

17.4 Command**

在实现 Layers (108)、Model-View-Controller (109)、Presentation-Abstraction-Control (111)、Active Object (212)、Enumeration Method (174)、Half-Object Plus Protocol (188)、Replicated Component Group (189)、Page Controller (196)、Front Controller (197) 或Application Controller (198) 的时候……我们在调用某个动作时，并不关心选择调用哪个动作。



通常，访问一个对象就是调用它的方法。但有时我们希望将请求发送者与请求的接收者解耦合，或者是对请求和接收对象的选择与何时执行请求解耦合。

在C/S的结构和部署中，将请求与其接收者显式地绑定在一起往往是有益的，但当请求本身比接收者是谁更为关键时，这种绑定也可能会导致过多的耦合。同时，在使用传统同步“调用/返回”方法调用模型的语言中，很难异步地调用一个方法。而且，内务操作如日志和撤销，在传统语言中也无法进行透明的支持。理想情况下，客户端应该能够发起一个请求，剩下的自然地发生。

因此，将发送给接收对象的请求封装在Command对象中，并为这些对象定义一个通用接口，来执行它们所代表的请求（见图17-6）。



图 17-6

当客户端需要发送特定类型的请求时，创建一个相应的Command对象。当其被调用时，Command对象被创建并初始化，并根据执行时接收到的参数控制相应请求的执行。



Command模式将调用操作的对象与处理操作的对象解耦，并将具体操作的选择与运行解耦。这种松耦合的关系确保了请求者不依赖于接收者的特定接口。因此，对请求接收者接口的改动不会涉及使用它的客户端对象。而且，将请求提炼为Command对象使得我们可以在应用中将请求作为第一类实体（First-Class Entity）来处理，这就使一些附加的请求-处理特征如取消（undo）和日志（logging）成为可能。

要实现Command模式，首先要定义一个Explicit Interface (163) 来统一Command的执行。该接口将会定义一个或多个执行方法，并根据需要接收参数。具体的Command实现该接口来处理特定类型的请求。在具体的Command初始化的过程中传入其执行过程所需要的状态，比如接收者对象或者方法参数等。Interpreter (258) 是一种特殊形式的Command，在这种模式中，一种简单语言的脚本被转换为一种可执行的运行时结构。

Command对象可以通过保留其所执行的行为的状态，来提供一种取消机制。但是，如果状态的数据量很大，或者难以恢复，Memento (242) 模式在执行Command前对接收者的状态进行快照可能是一个更简单、更轻量级的方案。

使用Composite (185) 结构支持创建和执行宏Command，将多个Command对象聚合起来，统一到一个单独的接口背后，并可以以特定顺序执行或回滚这些Command。独立的Command Processor (199) 根据发送者行为执行Command对象，可以提供额外的请求-处理支持，如多次取消/重做、执行计划和日志。

17.5 Memento**

在实现 Explicit Interface (163)、Introspective Interface (166) 或 Encapsulated Implementation (181) 的时候，或是应用 Observer (237)、Command (240) 或 Component Configurator (289) 等模式的时候……我们需要在参与者之间交换状态而又不破坏其封装性。



我们常常需要记录一个对象的内部状态。允许其他对象直接访问对象的内部状态，不仅会破坏其封装性，还会导致被依赖对象不必要的复杂性。

存储和读取对象状态在分布式系统中是比较普遍的。例如，当一个对象进入休眠期，持久化服务可能需要抽取其状态并保存，使对象在下一次运行时可还原之前的状态。但如果因此而保留对象的内部状态，我们就很难控制别的对象对该对象内部状态的访问，甚至是修改。由于缺少封装性，相关的对象可能会被绑定到某种状态表现方式上，这将导致我们的软件非常难于修改。

因此，在一个独立的Memento对象中对原对象的相关状态进行快照和封装，将该Memento对象传给需要使用原对象的客户端，而不是让客户端直接访问原对象的内部状态（见图17-7）。

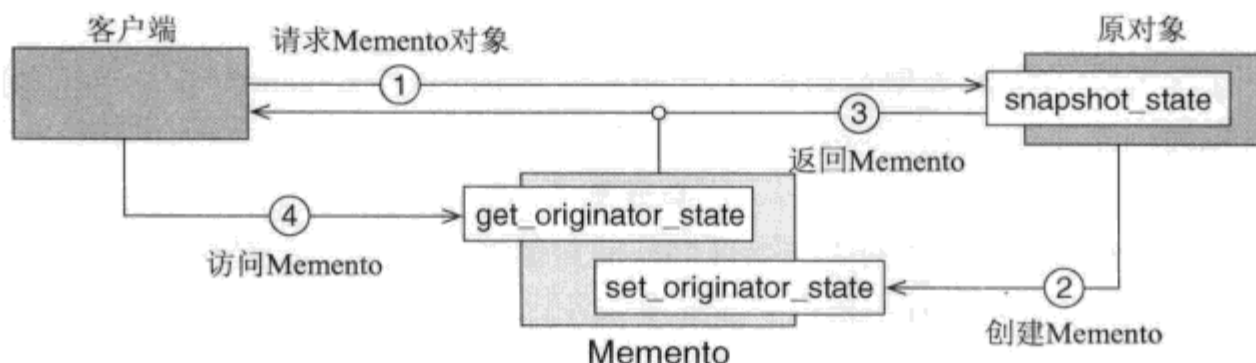


图 17-7

当从客户端返回，或是被状态恢复服务还原时，原对象可以从该Memento对象中恢复并激活其原先的状态。



Memento保留了原对象的封装性——依赖对象无法直接访问它的状态，但原对象自己可以。Memento允许这些依赖对象通过对发布对象的请求来取得原对象最近的状态，这些状态是经过包装的，不需依赖于状态的特定表现形式。

Memento内部存储的表现形式可以根据软件的不同版本自由地演化，而不用对依赖对象进行任何代码级别的改动。事实上，根据具体使用的编程语言和类设计技术，也有可能做到各个版本间的二进制兼容性。

大多数Memento实现提供了两个接口。对于原对象，它们提供了一个宽接口，以及setter和getter方法来初始化Memento对象并对其保存的状态进行访问。但原对象的客户端往往只能看到一个只读的窄接口，而不能修改Memento对象的状态。某些支持软件包结构的语言如Java和C#，允许包内与包外对象的方法有不同的可见性，这使得两种接口的分隔非常简单。C++可以通过友元、

继承以及用软件包组织源文件来支持不同的可见性。通过继承进行分离也可以帮助对宽接口和窄接口进行区别。

17.6 Context Object**

在实现Explicit Interface (163), Introspective Interface (166), Encapsulated Implementation (181), Interceptor (260), Interpreter (258), Lifecycle Callback (295)或Task Coordinator (296) 模式的时候……我们需要共享系统或调用上下文的信息而又避免引入全局的耦合。



在分布式系统中，松耦合是一个关键的设计目标。但在松耦合的系统中，有时候会需要共享一些与程序执行上下文相关的通用信息，如程序的外部配置参数、客户端进程状态及日志，它们跨越程序的不同部分和层面。

虽然一个程序的执行上下文范围往往较其单个部分的执行上下文更广，但全局变量和Singleton[GoF95]模式使得程序的任何部分都可以毫无限制地进行访问，也会造成不必要的耦合。全局变量和Singleton的实现也没有提供简单的后绑定机制，它们所基于的实现和实例往往在设计实现阶段就进行了硬编码，而无法在运行时进行绑定。相反，将良好定义的信息内容转化为多个独立的变量，将良好定义的服务转化为多个独立的操作，可能导致参数列表无法维护而又不稳定。

因此，将所需上下文中信息和服务封装在一个对象中。并将这样的对象提供给需要该上下文的操作、组件和层（见图17-8）。

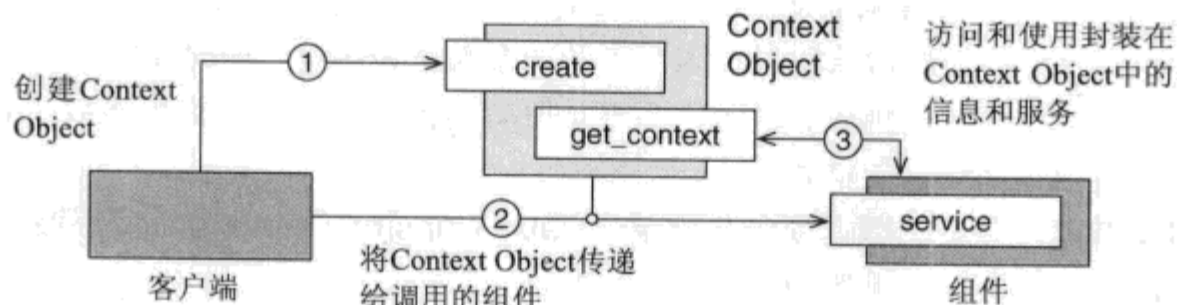


图 17-8

请求某组件的客户端创建一个Context Object，可以显式传入，也可以跟调用的其他参数一起隐式传入。接收请求的组件使用Context Object中的信息和服务来完成自身的执行。



使用Context Object模式，基本的运行时可置换性既可以显式地通过标准的参数传递机制来实现，也可以隐式地通过Thread-Specific Storage来实现。这使得我们可以在不改动配置或系统其他部分的代码的同时，对组件的执行上下文进行修改，还可以在不同的上下文中运行同一个程序，比如，采用多线程的方式。Context Object整体上作为一个稳定的单元，而不是每个独立的信息项或者服务操作，随着程序的运行，对Context Object的修改会逐步产生影响。

如果一段代码只是依赖于某个可传入的值，而不是某种执行上下文，我们就不应当向其传入Context Object，而是直接传入值本身。这时Context Object相对于值本身来说，只会引入过度的上

下文、混淆的含义和不必要的依赖。虽然执行上下文是一种横切特性，但这并不意味着它应该是全局的，就像一个全局变量或是全局类型那样。

Context Object引入了责任和依赖的反转 [Mar04]。它也可能引入控制流的反转，虽然并不总是这样：主要包含信息的上下文通常是被动的，但若是服务上下文是有关行为的，则Context Object模式控制流反转的特征往往就比较明显了。

17.7 Data Transfer Object**

在实现Model-View-Controller (109)、Presentation-Abstraction-Control (111)、Microkernel (113)、Reflection (114)、Shared Repository (117)、Blackboard (119)、Explicit Interface (163)、Introspective Interface (166)、Combined Method (172)、Whole-Part (183)、Master-Slave (186)、Transform View (201)、Observer (237) 或Table Data Gateway (323) 等模式的时候……我们需要对远程组件对象中的数据进行查询和更新。



具有状态的组件往往需要一个接口对其属性进行查询，有时还需要对其进行更新。每次远程通信都可能引入巨大的开销，因此，将每个属性的get和set实现为独立的方法是低效率的。

通过良好粒度的调用来对远程对象的单个属性进行访问，其开销是非常大的。而且，这类接口可能会带来一致性的问题，因为如果一个客户端需要一次查询多个属性，在多个调用之间，其他使用者可能会改动远程对象的状态，即便单个查询能够保证同步。同样，在客户端读取远程对象的过程中对其进行锁定，效率也是很低的。如果只涉及单个方法，Combined Method (172) 也许能派上用场，但同样的数据对象出现在多个方法的参数列表中时，如果参数列表发生改变，可能会导致重复和接口不稳定的现象。而且，并不是所有的语言都能支持输出参数 (out parameter) 的概念，所以也就很难简洁地支持查询。

因此，将所有可能需要的数据项合并到单个Data Transfer Object上，使用该对象来实现对属性进行集中的查询和更改（见图17-9）。

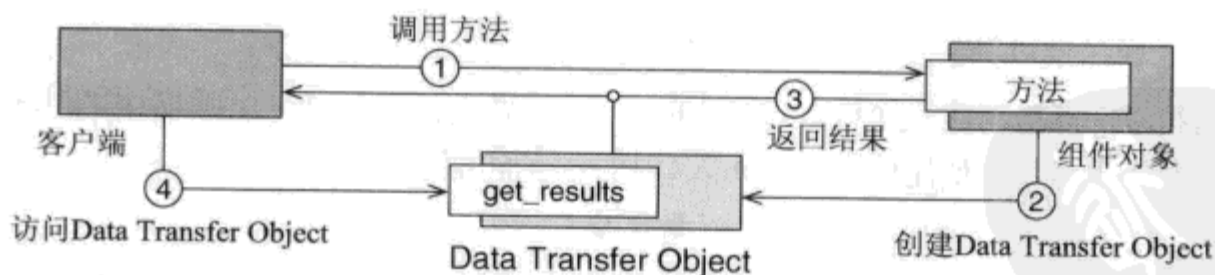


图 17-9

Data Transfer Object自己几乎没有什么行为，仅仅包含了访问属性需要的数据和查询条件，以及初始化和有选择地更新数据的方法。



Data Transfer Object能够更好地适应网络，因为只需要一次远程调用就能完成对属性集合的查询和更新。即使属性集合发生了改变，Data Transfer Object需要进行改动，远程对象的调用接

口依然可以保存不变。在非分布式系统中, 需要避免对组件对象的大量细粒度访问调用以及避免客户端依赖于组件对象内部的具体数据表示时, Data Transfer Object模式也能一展身手。

即使查询中并不需要所有的属性, 使用Data Transfer Object的一次调用的开销仍然往往较反复对所需的每个属性进行单独访问的总开销低。如果只有一部分属性需要更新, 调用者必须提供其他属性的现有值(这种方法可能需要一个额外调用来取回这些属性, 而这又一次增加了覆盖其他更新的风险)或是指出哪些属性无需更新。这既可以体现在Data Transfer Object中, 也可以体现在方法签名中。

Data Transfer Object需要使用某种方法来显示哪些值没有被设置。对于某些数据类型来说, 可以使用null来表示。但就原生数据类型(Primitive Type)来说, 可能需要额外的标识来体现。也可以用一个简单的标示集合来说明哪些值经过设置了, 并随着Data Transfer Object一起传送, 但这种方法可能容易出错, 用起来也比较笨拙。一种更弱类型化的方式是使用“名字-值”对组成的Data Transfer Object。

17.8 Message**

在实现Layers (108) 或Pipes and Filters (116) 结构, 或实现Messaging (129), Broker (137), Requestor (140), Invoker (142), Client Request Handler (141), Server Request Handler (249), Publisher-Subscriber (135), Half-Object Plus Protocol (188)及Replicated Component Group (189) 模式的时候……我们必须提供一种方式通过网络交换信息, 而又不引入对具体组件类型及其接口的依赖。

17

分布式对象像并列的组件一样协作, 调用彼此的服务并互相交换数据。但是, HTTP这样的网络协议只支持比特流这种最简单的数据传输形式, 并不能识别服务调用和数据类型。

我们往往需要以类型化和结构化的形式在网络中传输服务请求、调用参数、服务处理结果以及其他信息如异常消息。否则, 发送和接收消息的组件无法了解它们收到的比特流具有什么语义。

因此, 将需要通过网络传输的方法请求与数据结构封装到Message中: 比特流包含一个信息头, 定义了传输的信息类型, 其来源、目标、大小以及其他结构信息, 然后是负载(payload), 它包含了实际的信息内容(见图17-10)。

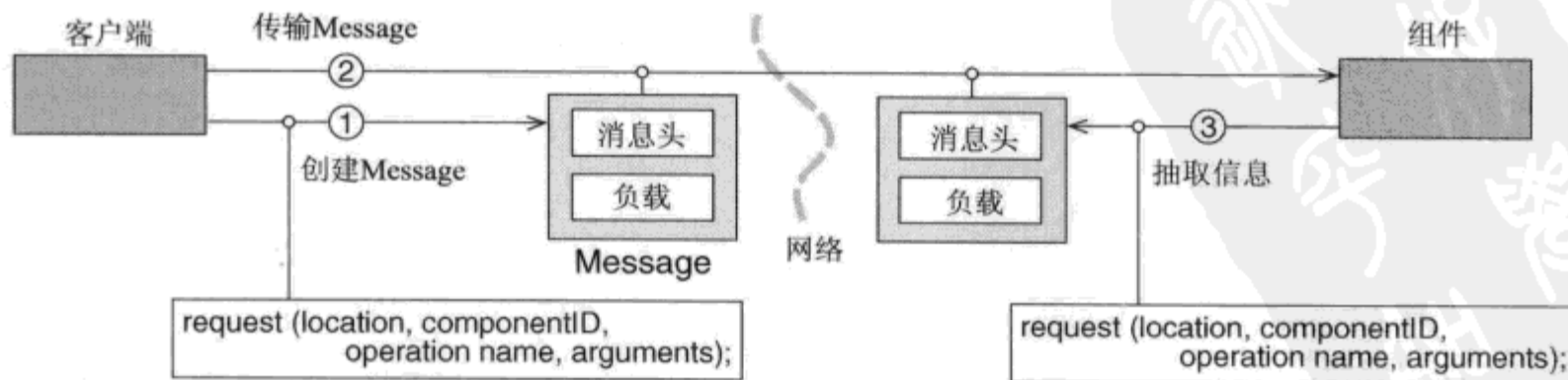


图 17-10

在发送者一侧，对远程组件的方法请求或数据交换都被封装到一个通过网络传输的Message对象中。接收者收到该Message对象后将其解封送，还原到原来的形式。



Message模式最大的好处是其包含的信息结构以一种“扁平”(flat)的形式来保存。而且，Message提供了对松耦合的支持：只要两者就共用的Message格式达成一致，发送者可以不依赖于接收者的具体接口。

虽然所有的Message应该基于一个通用的Canonical Data Model [HoWo03]，但不同的Message类型应该是可以区别的。Command Message [HoWo03]可以表示请求调用接收者的某个方法或子程序；而Document Message [HoWo03]可以将一个数据的集合传给接收者；Event Message [HoWo03]表示将发送者状态的变化通知接收者。

如果我们希望接收者组件对其接收到的Message进行响应，Message应该包含Return Address [HoWo03]（返回地址）。其响应也同样封装在一个适合的Message中，这就形成了一个Request-Reply [HoWo03]（请求-响应）的场景。消息头中的Correlation Identifier（相关标识）[HoWo03]基本上是一个Asynchronous Completion Token (155)模式的实现，它指明了响应Message的接收者。

如果传输信息的大小超过Message的最大长度，可以将其分割成多个较小的Message，以Message Sequence（消息队列）[HoWo03]的形式传送给接收者。如果封装到Message中的数据只在有限时间里有用，或者说有效，则需要在消息头中定义一个相应的Message Expiration（消息超时）[HoWo03]。

独立的Command Processor (199)可以帮助我们将特定Message（或是Message队列）转换为对接收者具体方法的调用，并能提供更多的请求处理支持，例如多重取消/重做、调度和日志功能。





一套可插拔的适配器
© Frank Buschmann

生命中唯一永恒的就是变化！这个来自于十七世纪法国作家Francois de la Rochefoucauld的箴言用来形容软件却是恰如其分。软件是一个“活物”。一个软件或者组件要想取得成功并能够被人们长期使用，最关键的是要能够适应新的环境、通过扩展满足新的用户需求，并且不断地改进和升级。本章列出的13个模式就是要帮助开发人员构建可适应、可扩展、可升级的软件。

现代软件的类型多种多样，而客户也是千人千面。有些应用是专门为某一个客户开发的，有些则面对一个相当规模的市场。有些应用虽然只有一个目标客户，但是如果这个客户经常有一些重复的业务，或者其他客户也在寻找相似的应用，我们就应当仔细地考虑如何定义一个通用的基础，并在此基础之上进行进一步的开发。尽管同一个软件系统或者组件可以为多个用户提供服务，但是每个用户往往会有自己的独特要求，而且这些要求可能在默认情况下并不支持。比如下面这些。

- 带外扩展 (Out-of-band extensions)。有时候客户会要求在系统或其组件的控制流中加入额外的算法或者服务。比如将系统与已有的应用或者某种监控或安全服务集成在一起。这

种带外扩展经常是与特定用户紧密相关的。

- 特殊的算法。即使所有的用户要求的服务是一样的，他们也经常在关键服务上要求采用特殊的算法，或者要求对算法进行裁剪。比如，业务信息系统必须考虑有关的税务和会计政策，而这些随着时间的推移经常会发生变化。同时根据国家、地区的不同，或者使用该系统的公司法定状态的变化，这些规则也会有相应的不同。
- 服务扩展和约束。客户经常会对系统中的某些服务要求特定的扩展或者约束。比如，有些客户会要求在执行某个组件的服务之前做一些预处理工作，或者在其后做一些后续处理。有些用户可能只需要组件的一部分服务，所以需要对服务进行限制。相反，有些用户则会要求对某些组件进行扩展，以支持更多的服务。
- 多平台支持。不同的客户可能会要求在不同的环境中运行我们的系统。所以，系统的组件应该可以移植到不同的操作系统、库、网络和硬件平台上。分布式系统的组件必须考虑在异质计算环境中的部署：组件可能会部署到多个平台上，并跨越多个网络节点。

要在不破坏基线架构和基础设计的前提下支持针对特定用户的配置、适配和扩展实属不易。更为糟糕的是，这种需要往往是在系统已经产品化之后提出的，而不是在开发或者装配过程中提出的。采用交互的、基于反馈的开发流程可以尽早澄清需求，但是对于此类的需求将其提前到开发阶段效果并不明显，因为这类需求的最主要的来源是系统作为产品应用之后的反馈。

然而，不幸的是，当前最常见的实践中，人们仍然是通过对系统做临时（ad hoc）修改的方式来处理单个客户的需求，而不是将系统最初安装完成之后的开发作为正常开发流程的一部分。从长远的角度看，这种做法并不合适。使用这种适配和扩展方式，散落在各处的修改和临时的解决方案经常会削弱甚至破坏原有的架构，从而使得代码基础越来越脆弱、杂乱，并难以维护。由于开发人员很难在整个系统的背景下去预见各种适应性的变化，也就使得他们很难取得高质量的实现和稳定的架构。于是原本通用的代码中充满了专门处理某种特殊情况或者专门为某个用户所做的修改。随着时间的推移，这些临时性的修改就会影响到系统架构的完整性，并最终导致产品再也无法处理新的需求和新的环境[BeLe76][Par94]。

为了避免遭受千刀万剐后缓慢走向死亡，我们的设计必须对其配置、适应性和升级能力做好充分的考虑。没有足够的约束和识别能力，我们就会患上“空想性泛化症”^①，无谓地把系统搞得过于复杂和臃肿[FBBOR99]。

这个名字是Brian Foote起的，专门用来指示某种类型的代码“坏味”。如果你听到有人在说“哦，我认为我们总有一天会需要具备处理这种事情的能力”，然后就想让系统能够处理各种特殊的情况，而不管有没有这方面的需求，那么这个人就算得上是“空想性泛化症”患者了。这样做的结果只是增加了理解和维护的难度。如果这些功能最后都用上了，还算值得，但是要是用不上呢，最终只会成为理解和维护的障碍，所以还是不要的好。

在稳定性和适应性二者之间要取得平衡是很困难的，首先要考虑的是系统的性质，尤其是它的部署和修改。对于一个封闭的系统，它可能只有有限的几个部署，关于修改方面的考虑就远比

^① speculative generality，在《重构》一书中意译为“夸夸其谈未来性”。——译者注

广泛部署的开放式系统少得多。为了保证系统在相应的部署条件下的稳定性和适应性，目前有很多设计方面的技术可以应用。比如，分析方面的技术包括*Open Implementation Analysis and Design*^① [KLLM95]、*Commonality/Variability Analysis*^② [Cope98]和*Feature Modeling*^③ [CzEi02]都可以用来帮助我们找出哪些是系统中稳定的方面，哪些是易变的方面，并将其隔离开。通用的设计技术——比如主动依赖管理（aggressive dependency management）——可以帮助我们在代码基础中将变化的部分隔离出去。而迭代和增量式生命周期（iterative and incremental lifecycles）等开发流程也让我们得以用基于具体用户体验的经验反馈来设计我们的系统，而不是根据某种设计理论闭门造车。

不论是何种情况，我们的目的都是为了给系统开发做一个稳定的设计，需要变化的部分应该是可扩展可改写的，而不需要变化的基础结构和行为应该是封闭的，这也符合开放-封闭原则[Mey97]。

我们的分布式计算模式语言包括13个支持配置、适应、扩展和升级系统组件的模式。每个模式用来应对本章开头所列出的一种挑战。而且，其范围并不局限于分布式系统，所有的这些模式在一切有适应性和扩展性要求的软件，或者在一个较长的时间内需要维护和改进的软件开发过程中都可以应用。

- Bridge（桥梁）模式 (255) [GoF95]将实现和抽象解耦合，它把一个对象分为抽象和实现两个部分，并使得双方可以独立地变化。
- Object Adapter（对象适配器）模式 (256) [GoF95]将一个接口转换成客户端要求的另一个接口。它使得本来由于接口不兼容而无法使用的类可以应用于该系统。它使用一个对象来将原有的类进行包装，从而确保该类对外是经过封装的。
- Chain of Responsibility（责任链）模式 (257) [GoF95]通过让更多的对象有机会处理请求，避免将请求的发送者与其接收者耦合在一起。接收对象被串成一个链，请求沿着该链依次传送直到某个对象对其进行处理。
- Interpreter（解释器）模式 (258) [GoF95]将语法以对象的形式进行建模，为简单的语言定义一个解释器。它将语法以可执行的方式表现出来，并用Context对象作为执行的参数来携带调用状态。
- Interceptor（拦截器）模式 (260) [POSA2]允许将与事件有关的处理透明地插入到框架中，当相应的事件出现时自动触发相应的处理流程。
- Visitor（访问者）模式 (261) [GoF95]主要用于操作包含不同类型的对象元素的结构。这些操作方法可以针对每个具体的类型而不同，调用这些方法不需要修改被访问对象的类型。
- Decorator（装饰器）模式 (262) 支持为对象动态地添加额外的行为。使用Decorator来扩展功能比做子类化更为灵活。
- C++中的Execute-Around Object（环绕执行对象）惯用法 (264) [Hen01a]定义了一个helper对象，其构造器和析构器分别在一段指令序列之前和之后执行某些动作，来确保正确性或异常安全性，同时可以减少代码重复。

① 可译为“开放性实现的分析与设计”。——译者注

② 可译为“共性与特性分析”。——译者注

③ 可译为“特征建模”。——译者注

- **Template Method**（模板方法）模式 (265) [GoF95]为操作定义了一个处理流程框架，它为其子类定义了一些步骤。这样子类就可以重新定义算法中的具体步骤的实现，而不会改变算法的结构。
- **Strategy**（策略）模式 (266) [GoF95]定义了一族同时变化的操作。每个变化体封装到一个对象中，而所有的变化体共享相同的接口。这样对这些行为对象的使用便可以独立于其变化体的实现。
- **Null Object**（空对象）(267) [And96][Woolf97][Hen02a]通过提供一个默认的“什么都不做”的对象，封装了缺少实际对象的情形。
- **Wrapper Façade**（包装外观）模式 (269) [POSA2]通过为非面向对象的API提供一个简洁的、健壮的、可移植的、内聚的面向对象的接口，来达到封装函数和数据的目的。
- **Declarative Component Configuration**（声明组件配置）模式 (270) [VSW02]使得组件可以指明它们想要以何种方式集成到容器的组件运行环境和容器本身之中，这样容器便可以自动地执行集成。

以上这些模式还不能覆盖系统配置、适应和升级的各个方面。它们所关注的主要集中在服务的内部。我们在第12章中所介绍的几个模式主要是关注于如何保护客户端，使其在服务的实现发生变化时不会受到影响。创建具体的服务配置和对具体的系统配置的运行时进行管理，则由第20章中所描述的几个模式来进行处理。

本章所介绍的这些模式的共同的主题是将组件的特定方面与其核心实现进行解耦合。它们只是在具体要解耦合哪一方面有所区别。当然，有些模式应对的是同一方面的问题，有时候它们是可以互换的。

Bridge和**Object Adapter**模式主要是将接口和实现进行解耦合，它们将客户端期望或者要求的接口与组件实际提供的接口提供一个映射。这种解耦确保了组件的实现可以独立地变化而不会影响到其接口和客户端。这两个模式之间的区别在于范围的不同。**Bridge**将同一个组件的接口和实现连接在一起，而**Object Adapter**则用于将不同的组件结合起来。

图18-1展示了**Bridge**和**Object Adapter**是如何集成到我们的分布式计算模式语言中的。

《设计模式》[GoF95]中所讲的**Adapter**作为这个层次上的一个单独的模式有点过于宽泛，它涵盖了一族与适配相关的模式的本质，其中也包括**Wrapper Facade**，但是它并未严格地定义一组动机和相应的解决方案。《设计模式》中包含了两个基本版本的**Adapter**——**Class Adapter**和**Object Adapter**，每个版本在不同的方面做了不同的取舍，并且具有不同的结构。其他的变体主要跟可插拔性有关，但这两种基本的形式，对于我们的讨论来说已经足够了。前一种是通过继承的方式来实现的，形成的关系是在类这个层次上的；后一种则是通过聚合的方式实现的，形成的关系是对象这个层次上的。后一种的缺点相对较少，应用也更为广泛，是我们的语言中最优秀的模式之一。

下一个模式**Chain of Responsibility**帮助我们将请求的发送者和接收者进行解耦合。这个模式将所有可能的接收者对象串在一起，发送者只需向串中的第一个接收者发出一次请求。请求沿着接收者链传递，直到某个接收者对其进行处理为止。因此，**Chain of Responsibility**为从客户端向请求处理器分派请求提供了非常灵活的解决方案。

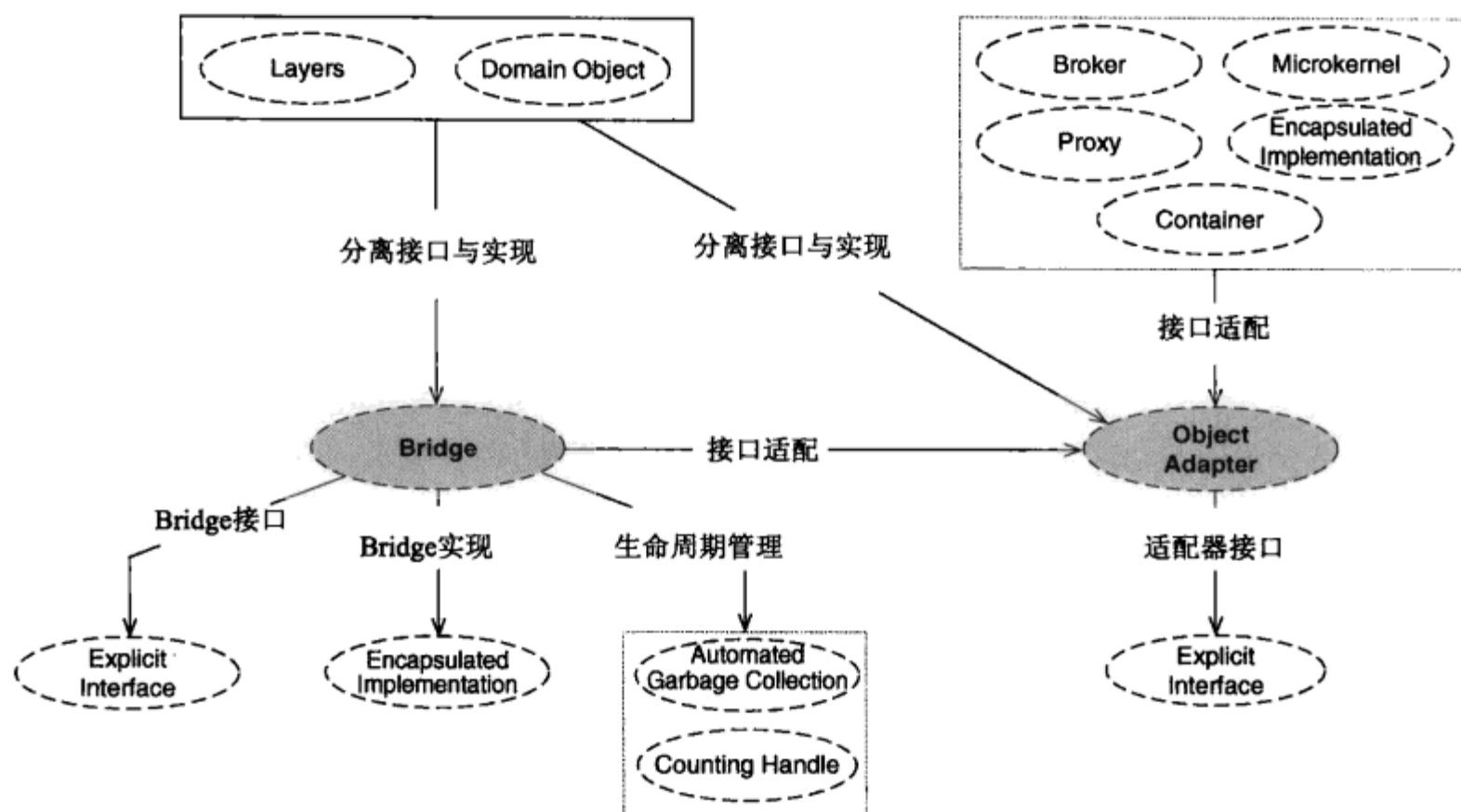


图 18-1

使用Interpreter模式实现的组件通过解释数据和脚本的方式来响应客户端的请求。这种设计的典型应用情形是，对组件的使用往往需要同时调用其上的多个服务，而又无法定义一个 Combined Method (172) 接口来规定一个调用顺序，或者很难确定常见的调用顺序。

图18-2展示了Chain of Responsibility和Interpreter模式是如何与我们的语言中的其他模式联系在一起的。

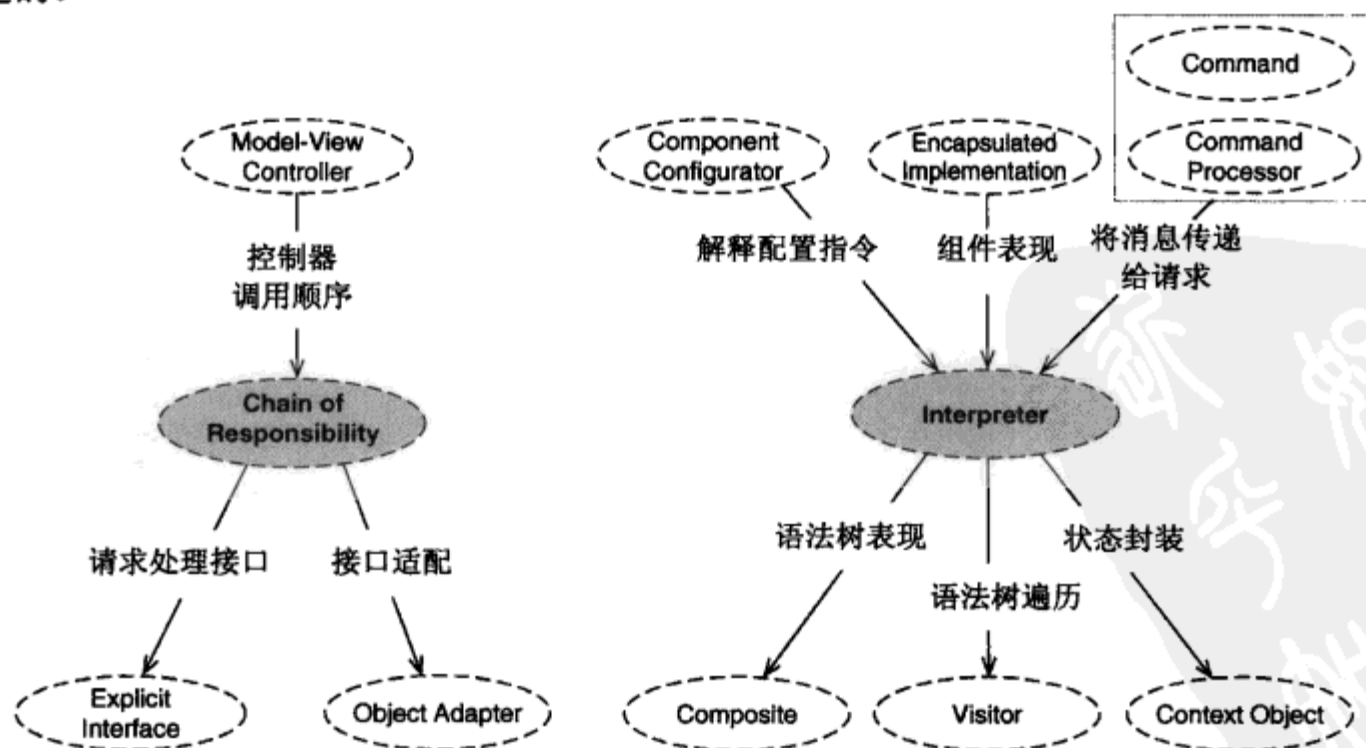


图 18-2

接下来的4个模式是有关如何对组件实现进行功能扩展的。通过Interceptor模式我们可以为组件的方法增加额外的控制流程，而通过Visitor模式我们将新的功能添加到一组相关的组件上面。Decorator模式支持为特定的组件添加新的方法^①，或者对已有的方法进行装饰，为其增加预处理和后续处理行为；而Execute-Around Object支持在C++对象的方法或者代码段前后执行预处理和后续处理动作。

图18-3展示了这4个模式是如何集成进我们的分布式计算模式语言的。

Execute-Around Object模式 (264) 有一个流行的名称叫做Resource Acquisition is Initialization [Str97]，缩写为RAII，这里我们使用的名字是取自*Executing Around Sequences*^② [Hen01a]。我们认为这个名字更能反映这个模式的本意，该模式的关键在于我们可以确保在对象生命周期结束时（而不是初始化期间）可以执行特定的行为。

Template Method和Strategy模式可以为组件提供可变的算法和行为。这两个模式的不同在于其实现的底层原理：Template Method使用继承的方式来解决，而Strategy则使用委托的方式。总的来说，对于实现变化的算法行为来说，委托的方式要更为灵活，所以Strategy的应用要比Template Method更为广泛。Null Object模式封装了一种特殊的Strategy：什么都不做。因此可以说Null Object为我们提供另一种形式的变化，算法从可变的变成了可选的，这是因为在某些条件下我们不能或者不应该执行该算法。这种变化形式通常称为消极变化 (negative variability) [Cope98]。

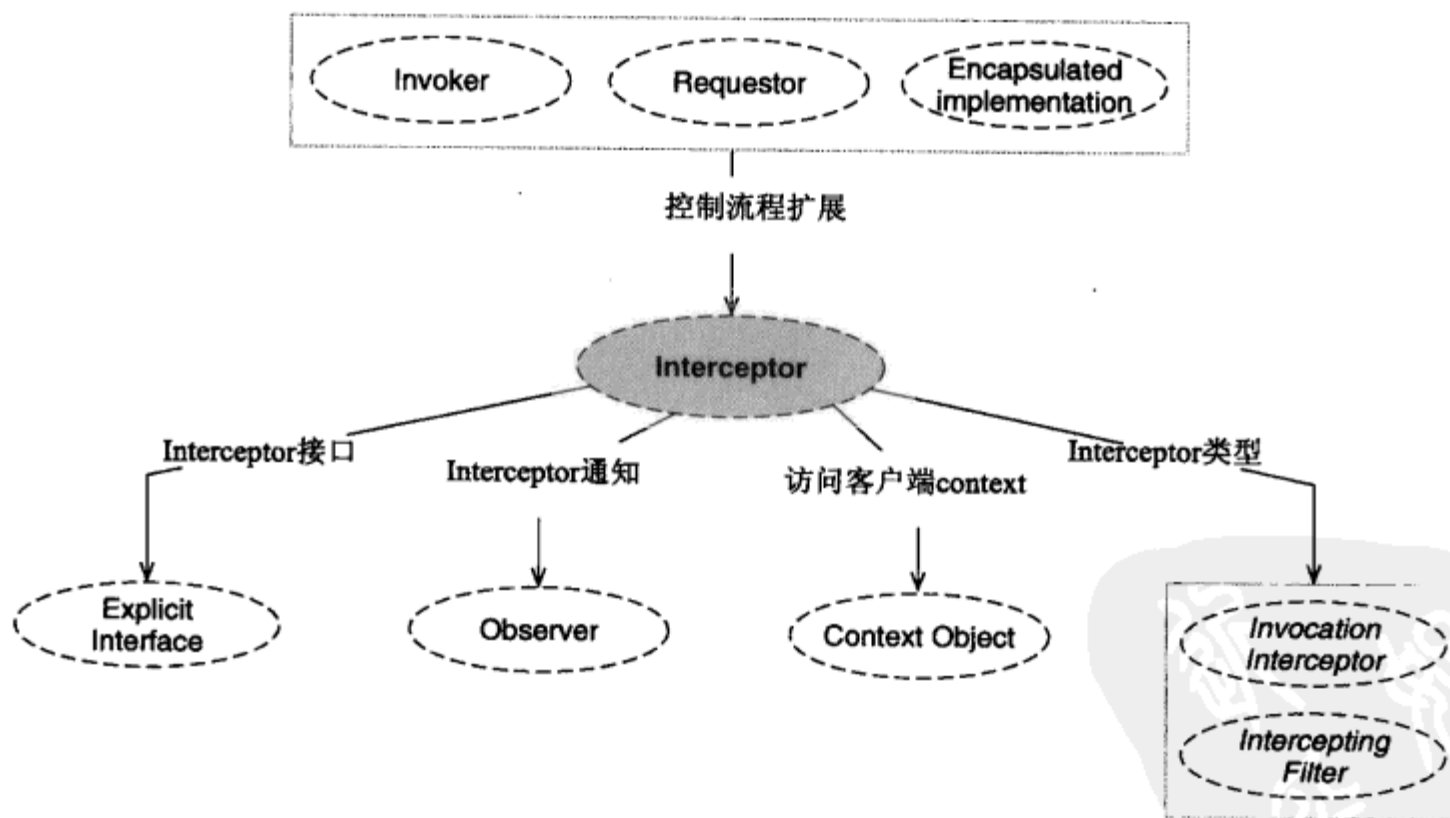


图 18-3

- ① 注意GOF的《设计模式》一书强调Decorator的实现“装饰对象的接口必须与它所装饰的Component的接口是一致的”，此处的实现与GOF的实现有所不同。——译者注
- ② 这篇文章中介绍了一组有关流程控制的C++模式，包括Execute-Around Object、Execute-Around Proxy、Execute-Around Pointer和Execute-Around Function。——译者注

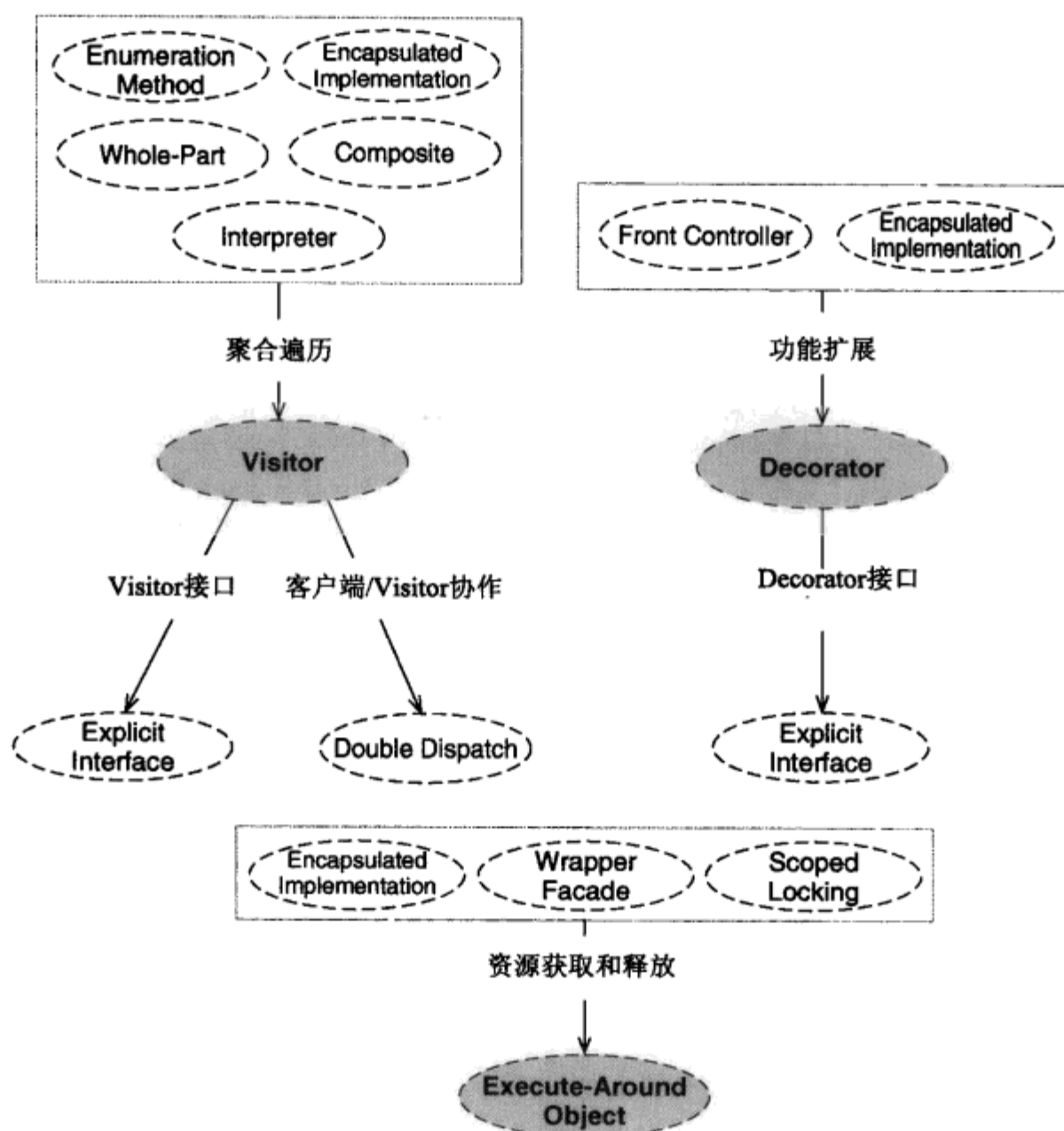


图18-3 (续)

图18-4展示了Template Method、Strategy和Null Object是如何与我们的模式语言集成在一起的。

底层的API所提供的函数往往是非面向对象的，比如操作系统API或者图形用户界面库，使用Wrapper Facades模式可以对其进行面向对象的封装，为其提供有一定语义的访问接口。Wrapper Facade的目的正是简化对底层API的访问，从而提高健壮性和平台独立性。从某种角度上说，Wrapper Facade和Object Adapter是非常相似的，但是它们所应用的范围有所不同，Wrapper Facade用来封装底层的API，而Object Adapter则仍然允许直接访问被适配对象的接口（见图18-5）。

本章最后一个模式就是Declarative Component Configuration，它用于解决如何将组件在资源和基础设施方面的需求传递给其宿主基础设施，以便为组件生成特定绑定。使用Declarative Component Configuration可以避免对此类绑定进行硬编码。

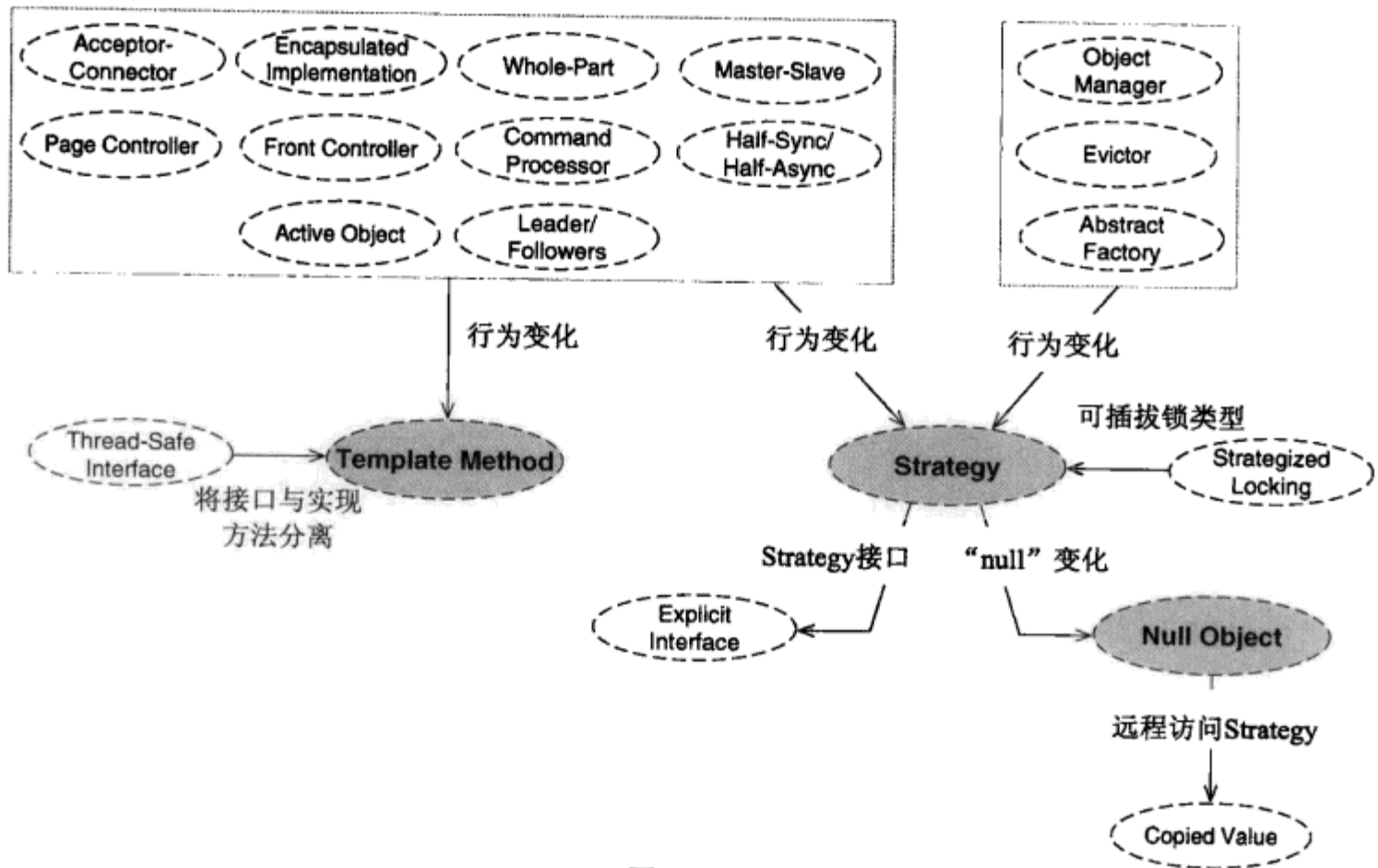


图 18-4



图 18-5

在其出处*Server Component Patterns*[VSW02]中, Declarative Component Configuration模式被称为Annotations。我们的语言中采用的名称则更为具体和明确,因为“annotations”通常被认为与专属于某个代码的元数据有关,Java中有一个标准特性就是这个名字。

图18-6展示了Declarative Component Configuration是如何与我们的模式语言发生关系的。

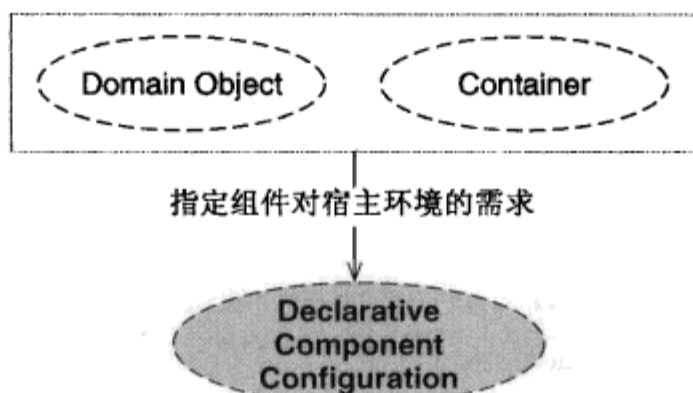


图 18-6

18.1 Bridge**

在实现Layers (108) 设计或者Domain Object (121) 的时候……我们需要允许对象的实现可以独立于接口发生变化。



一个接口可能会有多种不同的实现。这些不同实现之间的区别,有的是跟平台有关,有的则是运行时(runtime)选择。然而,使用继承来分离接口和实现,会把有关实现的选择暴露给对象的客户端。

继承是分离接口和实现的常用的方式:接口声明了对象的可见功能,实现则将接口声明的方法具体化。如果直接访问具体类,就会出现耦合,影响到系统的稳定性:客户端代码就会依赖于低层的实现类型。如果通过接口访问,对象的使用者就不会受到低层实现类变化的影响,但是它并非完全对实现没有依赖:在创建的时候,客户端必须决定选择哪个具体的低层类型,这同样会使得客户端代码变得混乱而复杂。

因此,将对象分为两个部分:一部分是抽象句柄,提供对象的接口,另一部分则是单独的实现层次结构(implementor hierarchy),为对象提供各种不同的实现(见图18-7)。

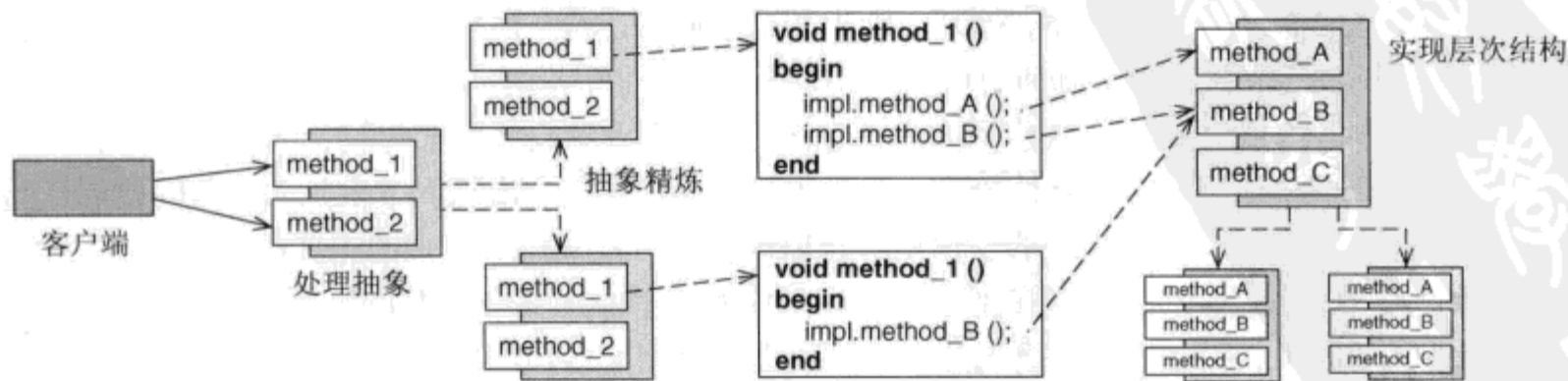


图 18-7

客户端只与句柄对象交互，这个对象由相应的实现类的实例进行配置。对句柄的方法调用传递给相关的实现进行处理。句柄也可以负责管理实现的生命周期。



使用Bridge模式，对象的客户端只依赖于接口，对实现的选择和实现的细节都被封装了。这样的好处是：客户端不会受到实现变化的影响。与之类似，如果对象接口发生了变化，只要新的接口可以通过Object Adapter (256) 等方式映射到已有的实现上，那不会对具体的实现产生影响。然而，这种严格地分离可能会带来性能上的损失，因为Bridge引入了一层额外的间接性。方法执行的时间越短，这种间接性带来的损失就越明显。

Bridge结构的两边都可以是具体的，也可以是一个类层次结构。如果是后一种情况，我们经常使用两个Explicit Interface (163) 来实现：一个用于抽象部分，另一个用于实现部分。具体的实现部分经常使用Encapsulated Implementation (181) 来实现。在实际应用中，我们经常采用动态连接的方式来为句柄选择合适的具体实现。

正常情况下，实现对象的实例是由某个给定的句柄独占的，如果这个实例是不变的（immutable），它也可以被共享。如果修改实现对象的成本可以推迟，则我们经常使用共享来进行空间优化。Automated Garbage Collection (307) 可以帮助我们防止实现对象被多个句柄实例意外析构（accidental destruction）。如果不支持垃圾回收，可以将句柄实现为Counting Handle (309)，以便跟踪实现对象被引用的数量。

18.2 Object Adapter**

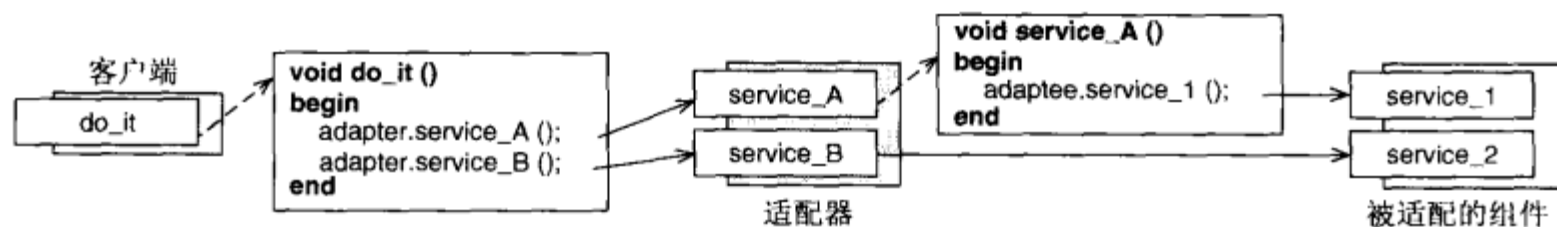
在实现Layers (108)、Domain Object (121)、Broker (137)、Microkernel (113)、Proxy (169)、Encapsulated Implementation (181)、Bridge (255)、Chain of Responsibility (257) 或者Container (288) 的时候……我们可能需要解决已有的组件所提供的接口和客户端所要求的接口之间的不匹配的问题。



对已有代码的重用可以在构建新的应用时提供帮助。然而，已有的类所提供的接口并不总能满足客户端的要求，有时候已有的类提供的接口会太多、太少，或者提供的接口的风格不符合期望。

改用原有类的接口固然简单，但是通常也会带来问题，因为客户端会被绑定在特定的实现和接口之上。接口的变化会影响到所有的客户端，而这对于重用来讲实在是不必要的副作用。而且，原有的接口可能不能正确地反映应用的真实意图，这就要求我们使用某种胶合代码来将二者连接起来，或者让应用代码适应已有的接口。前一种做法会使得客户端遭到（重复的）基础设施代码的污染，这一点非常讨厌，而且容易出错，难以维护，对于实现客户端的主要职责也没有贡献。后一种做法所产生的客户端领域模型从使用该组件的应用的角度来看不是那么理想。因为重用已有的代码是手段而非目的，所以重用的代码应该提供良好的接口，以便客户端可以按照自己的需要新的环境下使用。

因此，在组件和客户端之间引入一个单独的适配器，将组件提供的接口转换成客户端期望的接口，也可以说将客户端的接口适配成组件提供的接口（见图18-8）。



对适配器上方法的调用映射到被适配的组件，即被重用类的实例上面。组件返回的数据结构被转换成客户端所期望的数据结构。

Object Adapter使得客户端不必关心是否在新的环境中采用了已有的实现,也不会影响到客户端的接口。如果被适配组件的接口发生了变化,对系统的修改只集中在适配器里面。这些变化对于适配器的客户端来说是透明的。

适配器类本身可能是基于其客户端所期望或者要求的Explicit Interface (163) 的。适配器通常是通过组合来实现的。或者适配器在构造的过程中初始化被适配组件的实例，或者可以从外部传入一个被适配组件的实例，后一种做法中两者的关系更松散一些。在构造适配器的过程中传入被适配的实例，或者以类型参数的形式传入都是可以的。不论使用那种方式，最终对适配器的调用都会被传递给被适配的组件实例。

Object Adapter可能会引入一层额外的间接性和额外的对象创建。如果被适配对象的表现(representation)是嵌入在适配器的表现之内,我们就有机会消除这种开销。对于C++来说,我们可以在适配器内包含一个被适配对象类型的数据成员,然后通过调用适配器对象方法进行内联的方式减少这方面的开销。对于C#而言,被适配对象必须是struct类型的。对于其他情况,比如被适配对象是通过指针访问的,或者是Java等其他的语言,这层间接性和对象创建的开销则不可避免。

由被适配对象所提供的接口匹配到客户端所要求的接口，这个匹配过程越复杂，它所引入的代价就越高，这不仅包括运行时资源和性能方面的代价，还包括开发复杂性。如果这种开销达到无法忍受的程度，可以考虑干脆不使用该被适配对象，或者对其进行重构，以方便适配。

18.3 Chain of Responsibility*

在Model-View-Controller架构中，要把用户在视图中的输入传递给控制器……我们希望降低请求发送者和处理请求的对象之间的耦合。

有时候，在一个应用中可以处理某个客户端请求的不止一个对象，比如从输入设备收到的用户输入就是这方面的例子。然而，客户端通常对于哪个具体的对象处理自己的请求并不关心——它们只是关心应用正确地执行了自己的请求。

客户端可能也不关心请求的分发逻辑——决定哪个应用对象应该处理某个特定的请求——因此对象结构和请求分发逻辑上的变化不应当影响到客户端。然而，有时候将分发逻辑提取到一个单独的基础设施对象中是不切实际的。如果只有少数的几个应用对象可以处理某个客户端请求，或者客户端请求的分发逻辑相当的简单，这时候要单独去开发一个请求分发基础设施恐怕有点得不偿失。我们需要一种将客户端请求分发到接收者对象的机制，而且这种机制应该是简单的、高效的。

因此，将可以处理请求的对象串成一个串，让每个对象自己决定是否能够执行某个特定的请求。如果该对象可以执行当前的请求，它就执行这个请求，否则它就将请求传递给串中的下一个对象（见图18-9）。

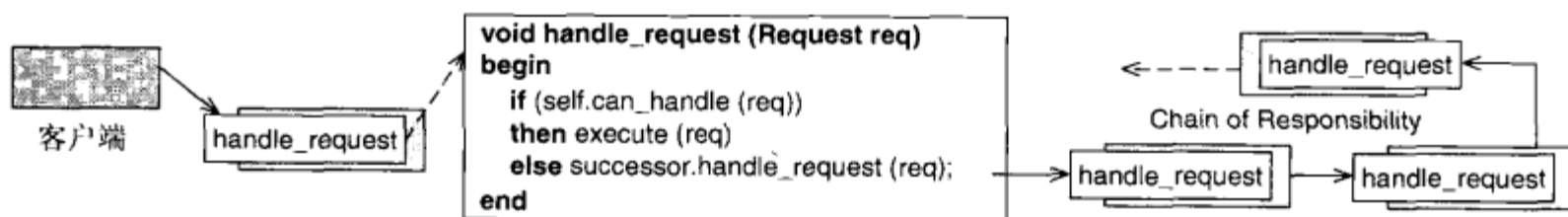


图 18-9

客户端首先将请求发送给Chain of Responsibility中的第一个对象。请求沿着Chain of Responsibility向后传递，每个对象检查自己能否执行这个请求。如果它可以执行这个请求，则执行之，并将结果返回；否则，该对象将请求传递给Chain of Responsibility中的下一个对象（实现了请求处理接口）；如果下一个对象不存在，则返回错误。



使用了Chain of Responsibility之后，客户端就不需要知道到底是哪个对象将处理它所发出的请求。这种灵活性使得我们可以动态地组合请求分发链，而不需要对客户端和链中的已有对象做任何修改。然而，Chain of Responsibility越长，它所引入的开销也就越大，因为随着跳数的增加，我们需要将请求传递更多次，才能开始真正的处理。而且，可能更重要的是，客户端无法确信Chain of Responsibility中是不是一定存在一个对象可以处理它们所发出的请求。

为了实现Chain of Responsibility，我们需要把Chain of Responsibility中的每个“下一个”对象的引用赋给“当前”对象，并且让所有的对象实现Explicit Interface (163) 以便接收客户端请求。我们依次调用各个对象上用于接收和处理请求的接口，直到某个对象能够处理该请求并返回相应的结果为止。如果对象的接口不符合Explicit Interface，可以通过Object Adapter (256) 来进行适配。

18.4 Interpreter

在设计Command Processor (199) 来接收请求消息的时候，或者设计Command (240) 和Component Configurator (289) 来接收基于脚本的配置参数的时候，或者更笼统地说，在设计参数化的Encapsulated Implementation (181) 的时候……我们需要一种机制来解释数据和脚本，并据此执行组件上的相应的服务。



很多问题的解决依靠的是解释而不是预先编译好的算法。比如，我们可以通过解释某个正则表达式来搜索匹配某个模式的字符串。要解释某个小的特定领域语言内的输入流或者数据模型，必须有一个语法表示，并且要能够实现根据语义执行。

像lex、yacc、Boost.Spirit和ANTLR等自动化代码生成语言处理工具支持对源语言的解析，但是它们并没有解决抽象语法树和代码执行的问题。对于小型语言而言——比如基于命令行的shell——我们不需要专门的间接表示，而是可以直接执行，甚至连一个专门的解析阶段都用不着。而有些语言具有自己的控制结构，其语法独立于上下文，这时候就不能简单地跳过事件解析直接执行了。

因此，引入一个解释器用于表示语言的语法和执行。解释器是一个整体-部分的结构，通常一个语法规则对应一个类（见图18-10）。

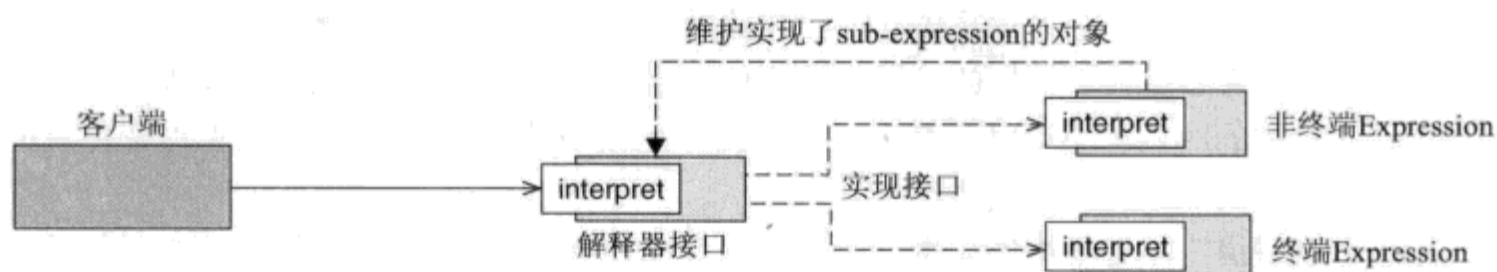


图 18-10

语言表达式（expression）之间具体的依赖关系组成了抽象语法树的基础。客户端需要进行词法分析和解析（parsing），该树结构形成了Interpreter模型的基础。叶子定义了语法中的终端表达式，非叶子节点定义了非终端表达式，非终端表达式由多个子表达式构成。子表达式可能是终端表达式也可能是非终端表示。

树的根是语言中最高层次的非终端表达式，为语言的执行提供了入口。根在所有的子表达式解释完成之后将所有的解释收集在一起，返回给解释器的调用者。



Interpreter模式为表示和解释小型的语言，比如结构化的消息和脚本，定义了一个直接而方便的方法，避免了更为复杂的表示模型。但是Interpreter模式并不适合于太复杂的语法表示，因为对于语法中的每一条规则都需要一个单独的类来实现和维护。在实践中，对于复杂的语法表示，我们通常采用自动代码生成语言处理工具这样的方式进行处理，它们可以将与解释相关的方面从语法中分离出来，在实际应用中可能更简单一些。

通常，我们使用Composite (185) 模式来实现语法树，管理这些具有相似类型的对象的结构和内部交互。Visitor (261) 通常用来完成对Interpreter整个对象结构的功能操作，比如语法检查、类型检查、生成输出，因为这样可以避免将这些功能分散在解释器的各个类内部。

在语法执行过程中，Interpreter结构本身是无状态的，执行的状态是通过Context Object (243) 传递给Interpreter的。

18.5 Interceptor**

在实现通信中间件中的Requestor (140) 或者Invoker (142) 部分的时候, 或是在实现Command Processor (199) 的时候, 或者, 更笼统地说, 在实现Encapsulated Implementation (181) 的时候……我们往往需要对组件或者框架的服务进行扩展。



对于一个框架来说, 其行为在怎样的环境和应用中需要做怎样的裁剪是很难预测的。本来核心服务是一个非常稳定的集合, 到新的环境中我们可能就需要对其中的某些特性和属性进行调整或者扩展。然而, 行为方面的修改是横切性质的, 而且几乎会关系到所有的对象。

例如, 通信中间件必须提供远程的IPC服务, 但是并不是所有的用户都要求对通信进行负载均衡, 也并不是所有的用户要求的安全策略都是一样的。以渐增的方式向中间件框架中添加此类服务并不可行, 因为这样会使得系统功能和代码随着时间的推移无限制地膨胀, 而用户(也包括开发人员)都得为那些极少使用的扩展付出代价。而且, 中间件框架最初的开发人员和维护人员在实现某个扩展的时候未必选择了最合适的方式。因此, 有些应用必须向中间件框架中集成一些额外的服务, 以满足其特定的需求。

因此, 允许用户通过使用回调接口——也称为“Interceptor”——来注册额外服务扩展的方式对软件框架进行裁剪, 当特定的事件出现的时候框架将自动地触发这些扩展(见图18-11)。

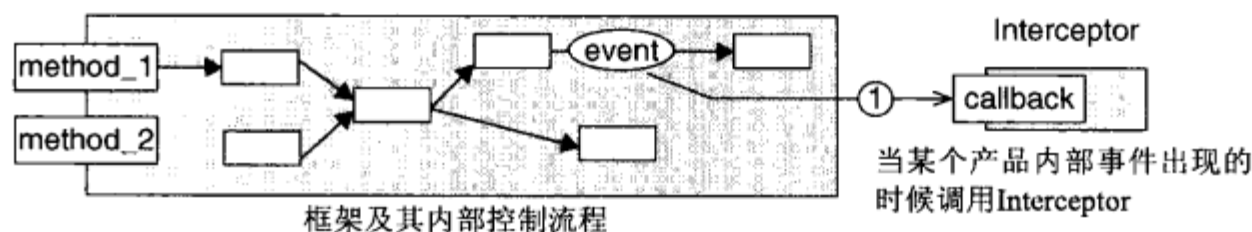


图 18-11

Interceptor实现了预定义的回调接口, 当相应的事件出现的时候以某种(专属于某个用户的)方式进行处理, 从而达到对服务的扩展。所有的Interceptor必须向框架注册, 以便相关的事件出现的时候框架能够通知到它们。在Interceptor接到通知之后, 开始执行其相应的功能。在Interceptor执行完之后, 控制流程返回给框架。



Interceptor基础设施支持对框架或者框架内的组件进行配置, 为其增加新的、未预期的功能——这些功能往往只有少数应用需要, 因此对于不需要这些特性的应用不会带来任何开销。框架只提供所有用户都需要的核心功能, 这样可以保持框架的精简, 从而提高其可用性和通用性。对框架和Interceptor的严格分离可以使得核心功能和服务扩展得以独立地变化和升级。

Interceptor模式所提供的高度灵活性也会带来一定的副作用。比如, 不管是不是有具体的Interceptor注册到框架上, 当相关的事件出现的时候通知Interceptor的基础设施都会执行。这意味着, 如果有很多拦截点和很多事件, 这种设计会招致相当可观的时间和空间开销。而且, 框架对于注册的Interceptor没有直接的控制, 而且也不了解它们的操作质量。因此很难为框架定义一个

严谨的规范，来说明它在Interceptor出现错误或者安全漏洞时的性能和行为。

要实现Interceptor模式，首先选择要注册的事件——用户希望在这些事件发生的时候执行带外扩展服务。比如在Object Request Broker (ORB) 中，客户端经常需要在对服务请求进行封送和解封送之前或之后提供事务和安全支持。将选择的事件分为两个集合：一个可读集合 (reader set) 和一个可写集合 (writer set)。在读集合内的事件出现的时候，用户只能获得有关这个事件的信息，而不能介入产品本身的控制流程，以避免对产品稳定性的影响。当可写集合内的事件出现的时候，用户可以修改产品的状态，这样可以支持更为复杂的服务扩展，但是这也有可能损害产品的状态，甚至导致错误的行为。将相关的事件划分成不相交的拦截组。比如，我们可以把ORB内处理请求发送的事件作为一组，因为用户在这些事件发生时往往希望执行同样的扩展行为。

对于每个拦截组，定义一个Interceptor回调接口，即一个Explicit Interface (163)，该接口为组内每个事件指定一个方法。具体的Interceptor从这些接口派生而来，用来实现具体的带外服务扩展。理想情况下，Interceptor应该在返回之前捕获并处理所有的内部错误，但是作为健壮的Interceptor模式的实现不应当做这类假设，而是应当做最坏的打算，否则很可能在拦截出现问题的时候就会遭遇故障。

在框架中出现了可拦截事件，框架会通过一个分发器将其通知给相应的Interceptor。我们可以用两个Observer (237) 来实现必要的通知链。框架本身是一个publisher，分发器向其进行注册，并在相应的事件发生时收到通知。同时分发器也是一个publisher，Interceptor会向其进行注册，当相关的事件出现的时候它们会被回调以执行其功能。在分发器内，实现一个合适的Interceptor回调策略，比如按照Interceptor注册的顺序，或者Interceptor的优先级。引入一个分发器要比让框架直接通知Interceptor好，因为这样可以保持产品的设计与实现独立于Interceptor的回调接口，同时也独立于Interceptor的注册和通知基础设施。

在框架回调Interceptor的时候，它可以通过Context Object (243) 来传递与出现的事件相关的信息。这个设计也使得Interceptor可以根据框架当前的执行状况调整自己的行为。

Interceptor有两种具体的类型，即Intercepting Filters[ACM01]和Invocation Interceptor[VKZ04]。Intercepting Filters允许组件上的服务请求在执行之前被拦截并进行处理。而Invocation Interceptor支持向跨进程的通信中注入可选的基础设施功能，比如负载平衡和安全。

18.6 Visitor**

在实现Enumeration Method (174) 和Encapsulated Implementation (181) 的时候，或者实现Whole-Part (183) 或Composite (185) 集合的时候，或者实现Interpreter (258) 的时候……我们经常希望实现一种服务，能够操作集合对象结构。



有些服务操作的对象结构是复杂的甚至是异质的。在一个拓扑树中用来收集状态信息或者进行查询的服务就是这方面的例子。将服务的实现分散在定义对象结构的各个类中，会使得设计不容易理解，难于维护和升级。

这些问题都是因为设计缺乏模块化。一个单独的功能被分为多个部分，每个部分处理一个对象类型，它的实现是与整个结构中所有对象的类横切（cross-cut）的。由于服务的实现并不是集中在一个地方，它的实现越是分散，我们就越无法看清它的整体意图。同时，由于缺乏模块化，类也变得更加膨胀，本来不属于其基本职责的功能被源源不断地加进来，这些功能的加入破坏了原有类的边界。换一种方式，将所有的行为放到一个单独的方法中实现，根据运行时的型别信息来执行不同的行为，也不见得好多少，因为这样会引入大量层叠的“if-else”结构。

因此，使用一个独立的Visitor类实现服务，针对对象结构中的每个类根据其型别的不同实现不同的行为。对定义对象结构的类进行扩展，每个类增加一个方法用来接收Visitor对象，并且在这个方法中根据自身的类型回调并执行Visitor的相应方法（见图18-12）。

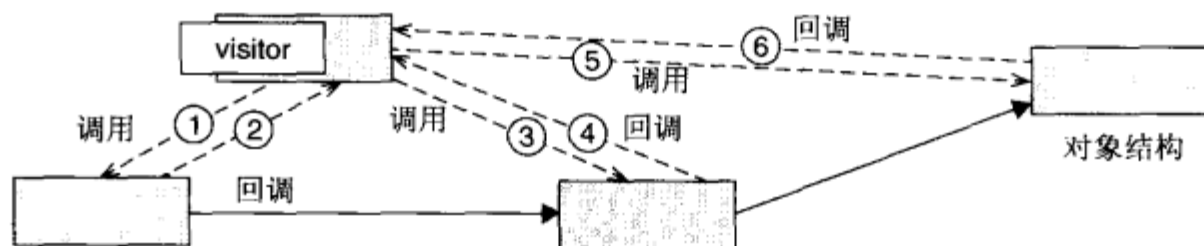


图 18-12

Visitor类为每个自己操作的对象类定义了一个方法。每个方法实现服务的一部分，这部分正是专门针对那个对象的类的实例的。将Visitor传入到要访问的对象结构中去，然后由结构中的每个对象回调针对自己类型的方法。调用的方法执行服务中相应的部分。



Visitor将服务进行了模块化划分，使得代码不再散落在各个类中，而是集中在一个地方——Visitor。这样提高了服务的算法和代码的易懂性和可维护性。Visitor基础设施也有利于维护设计的稳定性[Gam97]，避免出现不可控的变化，因为使用这个模式我们可以在不破坏原有设计和实现的前提下，向对象结构提供新的服务功能。稳定性是一个双向协约（two-way contract），只有在对象结构是稳定的时候才适合使用Visitor模式。向对象结构中添加新的类会对Visitor的层级结构产生涟漪效应，需要在每个Visitor类中均添加一个新的方法。

Visitor通常是通过Double Dispatch (238) 实现的。我们用Explicit Interface (163) 来声明所有的visit方法，每个方法对应于对象结构中的一个类，并将这个类的实例作为参数。具体的Visitor都是这个接口的子类，实现了某一项服务。在具体的Visitor中的每个visit方法实现了服务中和对应的类相关的部分。

在对象结构中的所有类都应该有一个accept方法，该方法以抽象的Visitor作为参数，并回调接收到的Visitor的visit方法。在回调时，它将被访问对象，即accept方法所在的对象，作为参数传递给Visitor。然后，每个visit方法在其回调时接收的对象之上执行其服务。

18.7 Decorator

在实现Encapsulated Implementation (181) 和Front Controller (197) 的时候，考虑到可扩展性和灵活性……我们经常需要为某个对象增加一些职责，但是又不希望影响它所属的类。



有时候我们需要对一个对象的方法进行动态的扩展，为其添加预操作和后续操作，比如数据压缩、记录日志、安全检查等。然而，如非必要，这些扩展不应该影响到同一个类的其他的实例。

我们当然可以将额外的行为直接集成到相应的类里面，然后根据客户端的需要来切换相应的行为，但这充其量只是个暂时的解决方案。随着时间的推移，类会被大量不同的选项所污染，就像一个“购物清单”一样，这些客户端需要这些行为，另外的客户端需要别的某些行为。问题是所有的客户端必须对所有的选项付出资源上的代价，这将带来性能上的损失和复杂性的提高。将这些插件式的行为隔离出来可以避免这种膨胀，但是也面临一种挑战：客户端通常不希望区分地对待所谓的“核心”对象和增加了额外行为的新对象。

因此，将原有的对象使用Decorator对象包装起来，Decorator对象的接口符合原有对象的接口。在Decorator对象中实现可选的、额外的功能，在这些功能执行之前或之后，将对接口的请求传递给原有的对象（见图18-13）。

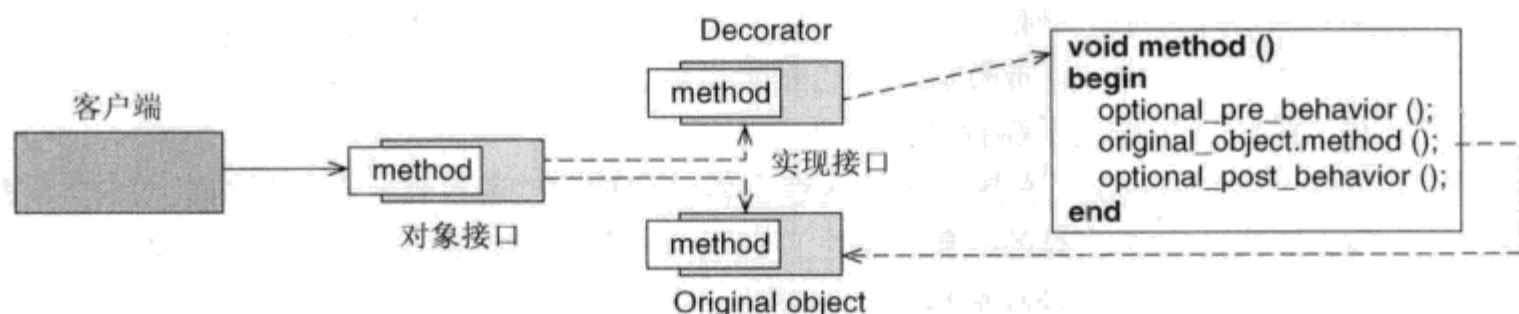


图 18-13

如果客户端调用Decorator上的方法，额外的行为和原有对象的核心行为都会被执行；如果客户端直接调用原有对象上的方法，则只有核心行为会被执行。



使用Decorator模式可以保持干净而简单的核心实现，而整个设计仍然是可扩展的。并且，这种扩展对于客户端来说是透明的。然而，因为客户端不知道自己访问的是原始的对象还是一个Decorator，它也就不会意识到自己的调用可能带来的成本。

要使用Decorator模式，需要把原始对象和它的Decorator组织在同一个类结构中。这个结构的根是一个Explicit Interface (163)，它定义对象的可见行为，也就是可以被装饰的行为。实现该接口的类有两种类型：一种是代表原始对象的类，其行为可以通过Decorator进行扩展；第二种类型则是所有的具体的Decorator。第二种类型中包含一个引用，指向一个实现了Explicit Interface的对象，这样一个具体的Decorator便可以“装饰”一个原始的类，或者另一个Decorator。这种设计使得我们可以使用多个嵌套的Decorator来对原始的对象进行装饰，从而形成一个链，每个Decorator可以扩展原有对象的某个部分。具体的Decorator从共同的基类中派生而来，实现了对原有对象的某些方法的预处理或者后续处理。

客户端编程时面向的是类结构的Explicit Interface。通过利用多态，我们可以对客户端进行透明的（动态的）配置，既可以使用原始的对象，也可以使用（嵌套的）Decorator。客户端调用最

外层的Decorator上的方法时，嵌套链上所有的Decorator都会被依次调用，最终调用将被传递给原始对象。

18.8 Execute-Around Object**

在实现Encapsulated Implementation (181)、Scoped Locking (227) 或者用来封装资源的Wrapper Facade (269) 时，考虑到适应性和可扩展性……我们需要在一系列C++语句前后成对地执行某些行为。



在C++中为了获取和释放资源我们经常需要成对地执行某些操作，即在某些语句执行前执行一些操作，在这些语句执行之后执行对应的另一些操作。这些前后执行的操作通常是跟语句块内的资源管理有关的，比如内存开辟、使用、释放。对于这样的序列在应用中进行显式的编码往往容易出错，而且很难做到异常安全，同时也容易引入重复的代码。

实践证明，人们经常忘记序列之后的操作。第一，开发人员必须确保在所有的返回路径中都会执行该序列，不论是解锁、释放资源，还是删除对象。第二，如果程序抛出了异常，后续操作就可能被跳过，这进一步提高了编程的复杂性。

因此，提供一个helper类，在这个类的构造函数中实现预处理操作，在其析构函数中实现后续操作。在栈上定义一个这个类的对象，确保它在指令序列之前构造，为其构造函数提供必要的参数，以便它可以在执行预处理和后续处理时使用（见图18-14）。

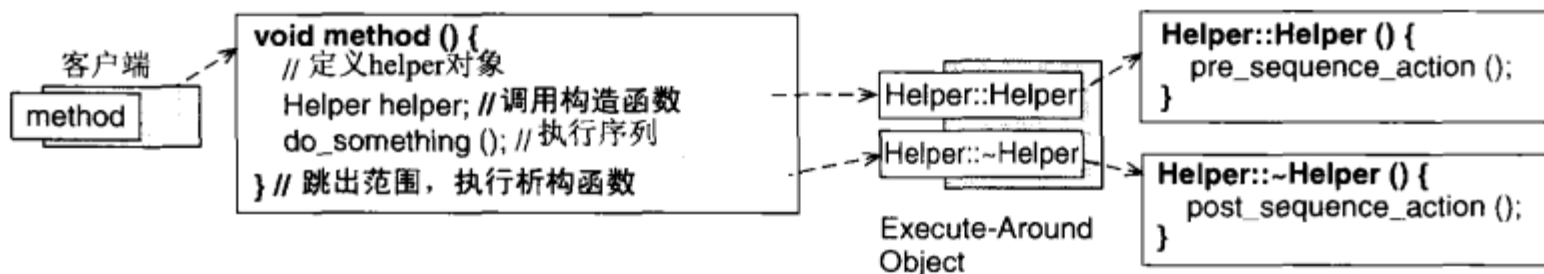


图 18-14

在C++中，程序通过调用构造函数来创建对象，并完成初始化，也就是说，构造函数是一个对象的“启动序列”。相反，在对象生命周期的最后，程序会调用析构函数来“关闭”对象。由于栈上对象的生命周期是绑定在封闭范围内的，所以我们可以确信析构函数一定会被调用，即使程序抛出了异常也没有影响。

上述机制保证了构造函数和析构函数对中间的指令序列形成包围之势。Helper类正是利用这一点，在构造函数中执行预处理操作，在析构函数中执行后续操作。如果helper对象需要调用其他对象上的方法，相应的对象和状态必须传递给构造函数。



Execute-Around Object模式同时实现了异常安全和对控制流的抽象，所以其代码重复相对较少，而且也不那么容易出错。栈在退出有效范围时展开（stack unwind），并调用栈对象的析构函数。不论是按照正常的控制流程退出范围，还是因为抛出异常而退出，栈展开都会发生。这样，

我们的清理工作就不会依赖于控制流程离开指令序列的方式。

如果后续的操作依赖于是否抛出了某个异常，我们可以通过`std::uncaught_exception`函数来判断调用析构函数的原因。但是，如果栈在创建`helper`对象的时候已经展开，那么`std::uncaught_exception`的返回结果就没有任何意义了。

Execute-Around Object依赖于两个关键的语言特性，而C++正好满足它们：一个是本地创建的对象可以确保在离开某个范围时会被析构；另一个是在对象析构之前会自动地调用析构函数。在别的语言中也有不同形式的执行包装行为的方法。比如，在C#中，要实现类似于**Execute-Around Object**的设计，可以使用`using`块，并让对象实现`IDisposable`接口。

18.9 Template Method*

在实现**Application Controller** (154)、**Encapsulated Implementation** (181)、**Whole-Part** (183)、**Master-Slave** (186)、**Page Controller** (196)、**Front Controller** (197)、**Command Processor** (199)、**Half-Sync/Half-Async** (209)、**Leader/Followers** (211)、**Active Object** (212) 和**Thread-Safe Interface** (224) 的时候……我们经常遇到有些对象它们的结构和行为核心是一样的，不同的只是行为的某些方面。



有些对象它们的结构和行为核心是一样的，而在行为的某些具体方面有所不同。为每个行为的变体提供一个完整的单独的类，对于其不变的核心来说，不但引入了大量的重复代码，同时也给维护带来了复杂性。

虽然说共用的结构和行为核心相对比较稳定，但是还是有升级的可能。一旦要升级行为核心，每个单独的类都要修改，这不但很无趣，也容易出错，如果各处的修改有不一致的地方，还会出现版本倾斜（**version skew**）。将对象不变的核心同其易变的方面分离开，并将其委托给另一个对象可以避免版本倾斜，但是这种自我完备的解法有些小题大做，因为这样就需要创建和管理两个单独的对象。

因此，为各种行为变体创建一个超类，这个类提供一个模板方法，实现了共用的行为核心。在这个模板方法中，将各种行为方面的变化委托给不同的挂钩（**hook**）方法，在子类中重载这些方法，实现各种不同的行为变体（见图18-15）。

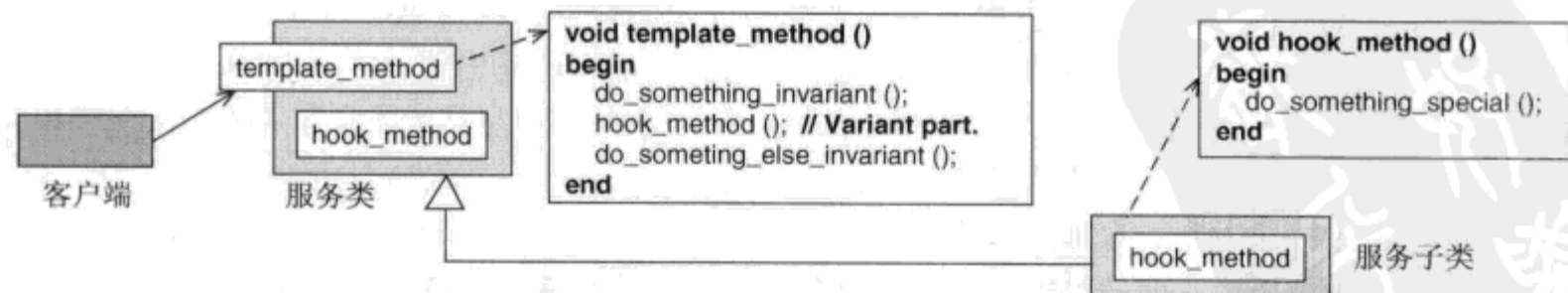


图 18-15

因此，使用该模式的客户端就只依赖一个类和一个对象。调用模板方法时，执行的是超类中共用的行为核心，然后由模板方法调用和执行挂钩方法，不同的子类对挂钩方法有不同的实现。



在Template Method设计中,所有的变体对象的结构和行为核心可以共享同一个实现,从而避免了代码重复和额外的维护工作。而且将服务变化的部分从其不变的部分隔离出来,使得二者可以独立地修改和升级。由于使用继承的机制来对两部分进行分离,这样它们可以直接共享同一个数据结构。

然而,如果服务的多个子类在某些挂钩方法上共享同样的实现,Template Method模式也会增加代码重复和维护成本。本来在对象不变的行为核心这个层次上要解决的问题,在对象变化的部分又被引入了。由于Template Method模式是类层次上的设计,而非对象层次上的设计,所以在行为的变体和超类之间具有较强的耦合。除非超类是一个非常强壮的设计中心(a strong design center),否则脆弱基类问题(fragile base class problem)将是一个不可避免的潜在的缺陷。如果组件为了自己的插件行为需要只暴露接口,则使用Template Method也是不合适的,因为它依赖一个部分实现的类。

在实现Template Method模式的时候,为了保证共用的行为核心不发生变化,我们可以将公开的(public)模板方法声明为非多态的。通常在私有的^①挂钩方法中,我们会实现默认的行为[Pree94]。如果默认的行为比较复杂,会使得该设计不那么稳定,而且在重载的时候也容易出错,因为子类的开发者必须非常小心,要知道自己重载了什么行为。

18.10 Strategy**

在设计Acceptor-Connector (154)、Encapsulated Implementation (181)、Whole-Part (183)、Master-Slave (186)、Page Controller (196)、Front Controller (197)、Command Processor (199)、Half-Sync/Half-Async (209)、Leader/Followers (211)、Active Object (212)、Strategized Locking (226)、Object Manager (291)、Evictor (305) 和Abstract Factory (311) 的时候……我们经常希望能够让多个对象共享同样的结构和行为核心,而在具体行为的某些方面允许有所变化。



有些对象的实现需要在一个或多个方法中根据不同的情况执行不同的行为。使用一个标志来区分这些情况,然后在实现的时候通过显式的方式来选择不同的行为,这种做法不仅脆弱而且伸缩性也比较差,属于封闭的解决方案。

这种基于标志的方式存在的主要问题是方法与标志的耦合,因为每个方法都会重复同样的switch结构。因此,各种典型的由代码重复而引发的问题都有可能出现:加入新的case时,在所有的case结构中都要重复相同的工作,更糟糕的是,我们可能会因为在某个地方漏掉而出现错误。这些方法通常都比较长,而且随着时间的推移还会越来越长。所以,这种方式不仅伸缩性差,而且是一个封闭的解决方案,每加入一个新的选项都必须修改已有的代码。

因此,将对象行为中变化的部分从定义服务的类中隔离出来,形成一系列Strategy类。在实现服务类的时候,将合适的Strategy实例插入到服务类的对象中,将不同的行为变体的执行委托

^① 在C++中我们一般将挂钩方法实现为私有的(private),而在Java中则一般实现为受保护的(protected)。更多信息可参考本书中文版wiki或者译者的博客。——译者注

给各个Strategy对象（见图18-16）。

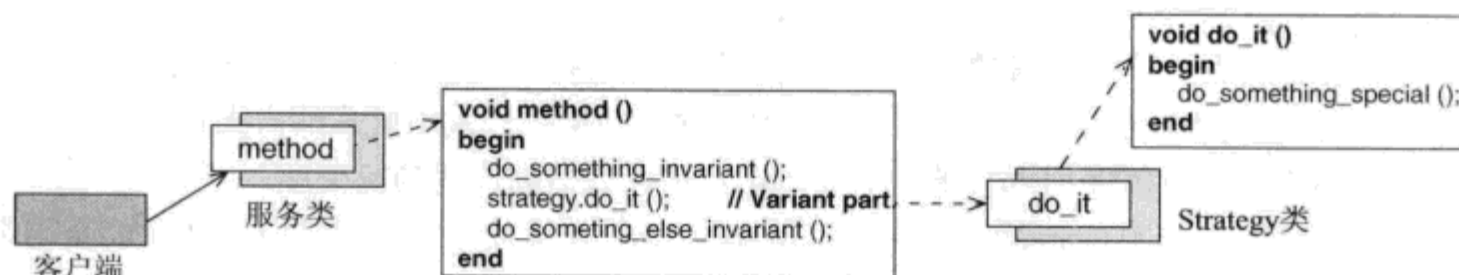


图 18-16

系统中既存在稳定的行为核心，也存在变化的部分，Strategy模式允许将容易变化的部分转化为参数的形式。



在Strategy模式设计中，服务类定义了应用代码，用来实现共用的行为核心，每个Strategy接口（包含一个或多个挂钩方法）用来实现行为中变化的部分。Strategy接口本身是一个Explicit Interface (163)，在服务类的实现中使用这些接口将服务中变化的行为委托给各个Strategy对象。具体的Strategy类实现了Strategy接口，为服务的某个行为变体提供实现。将行为中变化的部分同不变的部分清晰地隔离开，使得我们可以对其进行独立的修改和升级。如果Strategy类在执行自己的动作的时候会用到服务对象的某些或者全部状态，那么我们可以将服务对象或者一个Context Object (243) 的引用传递给它。

在为对象的服务实现行为扩展的时候，我们经常会用到Strategy模式。Strategy模式中的服务类的接口定义了扩展点，可插拔的扩展则实现了相应的扩展行为。正是因为这些，Strategy模式也被称为Pluggable Behavior[Beck97]。如果不需要执行任何行为，可以使用Null Object (267) 来实现某个具体Strategy，它为服务对象提供了简单而一致的工作方式，通常比在代码中直接使用null检查更为方便。

Strategy模式有两种具体的形式：基于实例方法的运行时多态和基于泛型机制的参数化（parametric）多态。比如C++中的模板和Java中的泛型就属于后一种情况。在基于运行时多态的实现中，通常由具体的Strategy类实现Strategy接口，提供行为的不同变体。而服务类在运行时使用不同的Strategy实例作为参数实现不同的行为。而基于模板的方式——也称为Policy[Ale01]，则是在编译期即确定采用哪种参数，从而可以减少对象创建和间接性（indirection）的成本，同时也为编译器内联提供可能性。如果有可能，我们可以在某些方面做一些灵活的取舍，包括型入侵、绑定时间（编译期或者运行时）和性能。如果无法对服务类对象进行运行时重新配置，就应该采用基于模板的方式。例如有的Strategy是跟操作系统或者硬件属性和API，或者其他系统环境紧密相关的，就属于此类情况。

18.11 Null Object**

在实现Strategized Locking (266)、Strategy (266) 的时候，或者笼统地说，在通过多态的方式来表达变化的行为的时候……我们应当考虑的一种情况是，对变化的行为给出一个“什么都不做”

的实现。



有些对象的行为只在某个其他的对象存在的时候才会执行。如果这个“其他的对象”不存在，那么这个行为或者什么也不做，或者使用某种默认值。如果我们采用显式的方式对对象的引用做条件检查然后进行分支就会引入大量的重复代码，并且会提高代码的复杂度。

反复地检查对象是不是null会打乱代码的流程，而且经常被忘记。很多时候，我们使用这个条件语句只是为了防止对空引用进行解引用：通常，如果引用为空，我们不会执行任何操作，或者简单地赋一个默认值。这些重复出现的条件判断会污染我们的代码，而且同时还会遭遇跟标志判断有关的所有缺点。所以，我们有足够的理由去除这些代码上的重复，更好地利用有关的语言机制。

因此，写一段“什么都不做”的代码：编写一个类，让它符合要求的接口，但是在其方法中什么都不做，或者简单地返回一个合适的默认值。在原来对象的引用为null的地方使用这个称作“null object”的实例（见图18-17）。

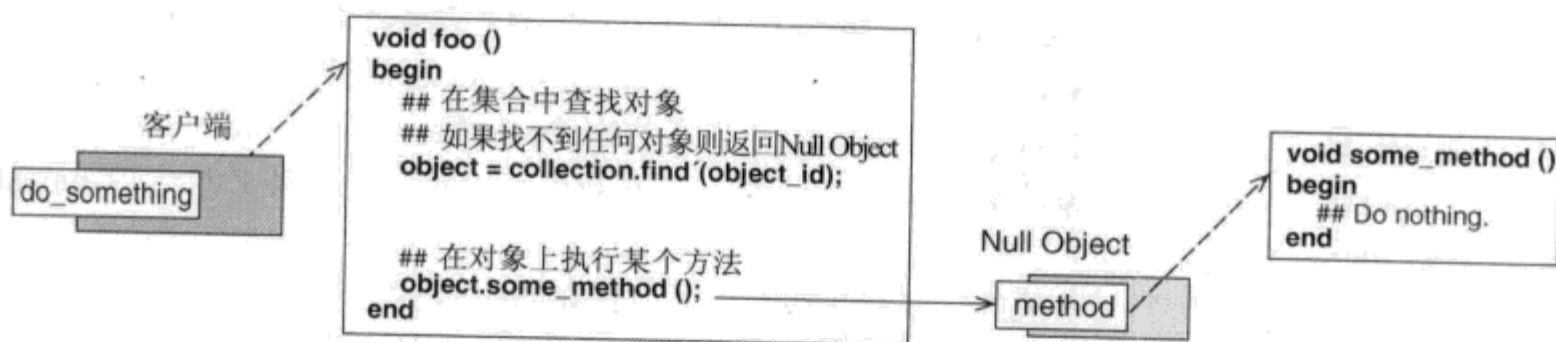


图 18-17

调用Null Object上的方法什么都没有做，所以也不会有副作用。在某些情况下，编译器甚至可以将调用的开销完全优化掉。



Null Object本身具有很好的封装性和内聚特性，它只完成了一项任务，就是什么都不做，而且完成得非常圆满。我们将“什么都不做”具体化为一个对象，并且对其进行封装，消除了额外的、重复的条件判断，而这些条件判断本来就不属于代码的核心逻辑。这使得我们更容易对付这种对象缺失的行为。选择和变化通过多态和（接口）继承的方式表现出来，避免了使用过程化的条件测试。Null Object设计移除了重复出现的条件判断语句。它利用了语言的封装和多态的机制，将决策机制对象化，并隐藏起来，对象关系从可有可无变成了一一对应，这使得我们可以统一地、一致地使用我们的组件。

然而，我们也不能不分青红皂白地滥用Null Object替换所有的空引用。如果对象缺失对于代码逻辑来说非常重要，而且相应的功能行为有较大的变化，就不适合使用Null Object。我们必须确保对这些可能为空的引用的访问和使用进行合适的封装。有时候，如果允许在多个客户端之间传递Null Object反而会把接口搞得比直接使用空引用更复杂。

根据定义，Null Object是无状态的、不变的（immutable），因此它一定是可共享的，而且是

线程安全的。但是对于远程对象来说Null Object的扩展性较差，因为方法调用是跨网络执行的，而且总是会对其参数进行赋值，这些都会带来严重的开销，而且也会引入额外的出错点，也对不起它们本有的“什么都不做”的优良品质了。在分布式环境中，Null Object应该以Copied Value (230)的形式传递，透明地复制相对于透明地共享更为适合这种环境。

18.12 Wrapper Facade**

在实现Broker (137)、Client Request Handler (143)、Server Request Handler (144)、Reactor (150)、Proactor (152)、Application Controller (154)、Model-View-Controller (109)、Encapsulated Implementation (181)、Half-Sync/Half-Async (209)、Future (223)、Strategized Locking (226) 和Scoped Locking (227) 等模式的时候……我们希望能够以一致的、健壮的、可移植的方式访问基于函数的底层API。



在应用中经常有些代码会用到层的非面向对象的API所提供的服务。如果在开发的时候直接使用这些API会使得代码难以理解。同时它对于可测试性、可移植性和长期的稳定性来说也不是一个好的选择，因为当时选择的平台也许明天就不合适了。

直接使用基于函数的底层API（通常使用C编写）会带来很多问题：一方面会引入大量的重复代码；另一方面由于过于关注API的细节而导致容易出错。同时这些代码的可移植性也比较差，甚至在同一个平台上版本发生了变化也可能导致代码无法正常工作。同一套API中不同的函数之间的关系也不明显。直接针对这样的API进行编程，会使得同样的代码散落在应用的各个角落，要想换用其他的解决方案将变得非常困难。

因此，避免直接访问基于函数的底层API。将一组相关的函数包装在一起，组成一个单独的、内聚的Wrapper Facade类（见图18-18）。

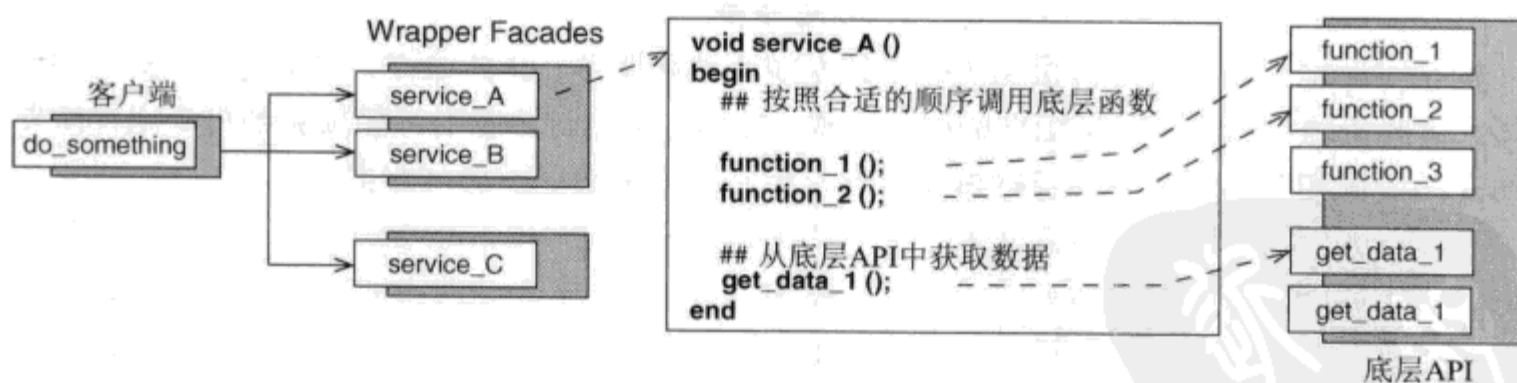


图 18-18

在执行客户端请求的时候，对Wrapper Facade上方法的调用会按照预定义的特定顺序传递给相应的API函数。



Wrapper Facade为访问系统API函数，比如操作系统和GUI库，提供了简洁而健壮的类。访问底层API的代码是经过良好封装的，所以访问代码不需要重复。Wrapper Facade对于提高应用的可

移植性也有帮助，因为即使底层的API变化了，wrapper的接口仍然可以保持稳定。Wrapper Facade是一个类，因此它可以在产生式编程（generative programming）模型中作为可插拔组件的基础[CzEi02]。

在待封装的API中，进行合适的抽象，定义它们之间的关系，这可能跟原始API中所做的声明有所不同。将一组相关的函数和数据结构组合在一起，构成一个独立的Wrapper Facade。API所产生的错误应该根据目标语言的风格进行处理——这也可能与API本身的风格有所不同。比如，用C语言编写的API通常是通过返回值来表示出现错误，而在面向对象语言中则使用异常来表示。在用C/C++编写的UNIX上的遗留系统中，Thread-Specific Storage (228) 设计可以支持基于单个线程的安全保持（safe retention）和错误处理。为了支持Wrapper Facade的健壮性，最好对资源获取和返回进行自动化处理。例如，在C++中可以使用Execute-Around Object (264) 实现。

Wrapper Facade设计应该保证对底层API的使用简单而容易，但是也应当支持复杂的应用情形。因此，Wrapper Facade的接口可能需要一个“安全舱口”，以便使用者可以直接访问经过包装的底层API。这种设计当然是做了妥协，但是它可以避免为每个特殊的情形修改Wrapper Facade，那样会导致接口过度膨胀，难以使用。在对底层的访问中，使用最多的就是用C语言编写的API。使用C++可以对其简单地进行语言内的包装，但是Wrapper Facade模式则是更为通用的方式。在Java或者Ruby这些与C之间有着更明显差别的语言中使用C当然更是如此。在这些情况下，如果只对底层的API做简单的包装——比如用整型来表示句柄，将函数导出成static的方法——相对于Wrapper Facade设计来说内聚性要差得多，后者显然更为合适。

18.13 Declarative Component Configuration*

在实现Domain Object (121) 和Container (288) 的时候……我们必须告诉应用的宿主环境，怎样处理某个特定组件的技术需求，比如其事务处理和安全需求。



运行时环境为应用的组件提供了必要的系统资源和服务，比如线程、网络连接、安全和事务处理。这些资源和服务限定了组件的执行方式。然而，运行时环境不可能预测到每个组件所要求的特定的资源和专门的服务，包括每个组件希望怎样使用这些资源和服务。

所以，运行时环境需要知道这些信息，以便能够正确地管理这些资源和组件。将资源和专门的服务需求用硬编码的方式写死在组件的实现中并不现实，本来功能方面和技术方面是各自独立的，这种做法却将二者混在一起了。要想独立地修改任何一方都非常麻烦，想要通过修改组件的实现去处理这些问题必须手动修改，代价不菲。每一次修改都会产生一个新的组件版本，这又会带来维护和管理上的问题。

因此，为每一个组件设计一个专门的Declarative Component Configuration，它可以告诉运行时环境组件需要哪些资源和服务，以及组件会怎样使用这些资源和服务（见图18-19）。

在部署组件的时候，将Declarative Component Configuration传递给运行时环境。运行时环境根据Declarative Component Configuration中的规范进行自我配置，以便能够正确地管理每个组件。

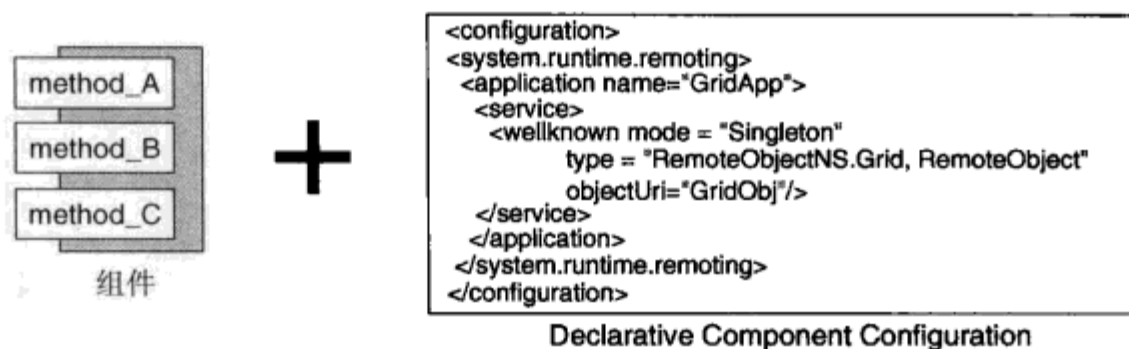


图 18-19



Declarative Component Configuration告诉应用的运行时环境如何管理其中的各个组件，运行时环境据此对每个组件的需求做出响应。这个模式可以提高组件的灵活性，因为组件可以指定其所在环境的某些方面，而不是只能被动地依赖于当时的环境，或者需要开发人员为每个组件手动写死其运行环境。将**Declarative Component Configuration**从组件接口和实现中分离出来，可以简化对组件的系统资源和服务需求规范的修改，因为这些不再会涉及对组件本身的修改。

Declarative Component Configuration通常会包括组件的名字、资源需求、对其他组件接口的依赖、所需的安全和事务支持、线程模型和各种服务质量参数 [VSW02]。这些信息通常都是通过配置脚本的形式提供，比如XML文件[OMG02][MaHa99][Ram02]。





布里斯托尔的人行横道显示牌
© Kevlin Henney

系统中的有些对象本质上是由状态驱动的：所有方法或者大部分重要方法的行为根据其当前状态的不同而不同。通常这种对象的生命周期最好实现为状态机的形式，从而可以对其模态行为（Modal Behavior）进行显式建模，并加以控制。本章给出了3种模式以支持状态机的实现，并折中考虑了各种方案的复杂性、性能、内存使用，以及内部和外部控制状态改变等因素。

有多种方法可以用来实现对象的状态驱动（State-Driven）生命周期。有时对象在方法控制流中使用简单的标记变量或者条件语句就足够了。然而，有时候对象的许多——甚至全部——方法在不同的对象状态下行为完全不同。通常这时其生命周期会用状态机进行建模，在实现中开发人员会面临很多选择，而有些设计会引入不必要的复杂性。下列因素会影响到我们的选择并决定了解决方案的结构。

- 减少条件变量。用大量的switch语句和很长的if...else if层叠在一起来表示对象生命周期中的状态行为通常是不合适的，因为它们会引入额外的复杂性。条件语句稍微一多就会变得难以管理，并且会导致多个方法中出现重复的条件结构。于是组件的方法（method）

会与某个特定的状态变量耦合在一起，从而变得非常难于扩展。

- 模式 (mode) 之间的依赖。有些对象的状态彼此完全独立：其方法不对公共数据结构进行操作，并且从一个状态转换到另一个状态不需要或者只需要很少的状态间上下文信息。另一些对象的情形正好相反：不同状态共享并操作一套公共数据和上下文信息。对象状态机的设计应当反映其模式间的相互依赖。将独立状态耦合得太紧密会降低其独立性，相反，相互依赖的对象如果耦合过松则会降低性能并增加资源管理的开销。
- 模式(mode)可见性。很多情况下客户端并不关心对象的当前状态，它们只希望在任何状态下对任何方法的调用都有适当的行为。管理状态和状态改变最好对这些客户端是透明的。在另一些情况中，对象客户端的行为依赖于这些状态，而对象自己则应当独立于这些状态模式。

本章介绍的3个模式 (pattern) 中可以应对上面给出的不同因素，为开发具有强模态行为的对象提供解决方案。

- Objects for States(状态对象)模式 (274) [GOF95] [DyAn98] 将对象一分为二，把对象模式相关的行为和一般的实例数据分离开。持有数据的主对象将方法调用转发给模式对象，而模式对象是代表对象状态的类结构的一个实例，每个类代表特定状态的行为。
- Methods for States(状态方法)模式 (275) [Hen02c] 将对象所有行为实现为单一类中的内部方法，而不是遍布在多个类中。采用方法引用分组的方式定义对象在某一特定模式下的行为。
- Collections for States(状态集合)模式 (276) [Hen99] 将对象的状态外部化 (externalize)，将对每个状态的关注表示为包含处于该状态的所有对象的集合，状态转换就成了在集合间的切换。

Objects for States模式通常被称为State模式 [GOF95]，而我们使用的名称是为了清晰和类比。Objects for States这个名字强调了解决方案的结构，而State这个名字强调的是问题。Methods for States和Collections for States使用了相同的命名风格，强调了其意图的相似性和结构上的区别。

在实现模态行为时下面的一些考虑影响了3种模式中究竟哪一种是最合适的。

- 模式可见性。Objects for States和Methods for States在模态对象内部实现状态机，因此对客户端是透明的。客户端“仅仅”调用方法，由对象根据其状态来做“正确的事”[Hearsay02]。因此Objects for States和Methods for States对封装特定工作流的对象实现最合适。相反，Collections for States在模态对象的外部——客户端的内部实现状态机，对象本身并不能意识到它的状态。这种做法乍一看有些奇怪，但其实不然。例如，对垃圾收集器 (Garbage Collector) 而言，那些被其他对象引用的对象不能删除，而未被引用的对象则可以删除。对象本身并不关心其状态，很明显它们也不应该关心。类似地，在支持多次撤销/重做的机制中，只有已经执行的请求对象才可以被撤销，只有被撤销的请求对象才可以重做。同样，请求对象本身通常并非像客户端那样关心这种状态相关的信息，这时应当采用Collections for States来实现。
- 模式独立性。有时对象的状态机包含彼此完全独立的状态而不共享行为或数据结构。这

种状态机中的每个状态应当被严格分离开，以避免状态机实现中过多地引入结构上和逻辑上的复杂性。Objects for States通过将每个状态封装在单独的状态对象中来满足这一需求。类似地，Collections for States为状态机中每个状态引入单独的集合。另一方面，如果许多状态共享行为和数据，则封装每个状态将会因为重复代码以及在状态间传输共享数据而引入空间和性能开销。为此，Methods for States提供了一套共享的方法和数据结构，并通过它们来组成特定状态的行为。

图19-1描绘了有关模式行为的3个模式是怎样联系在我们的分布式计算模式语言中的。

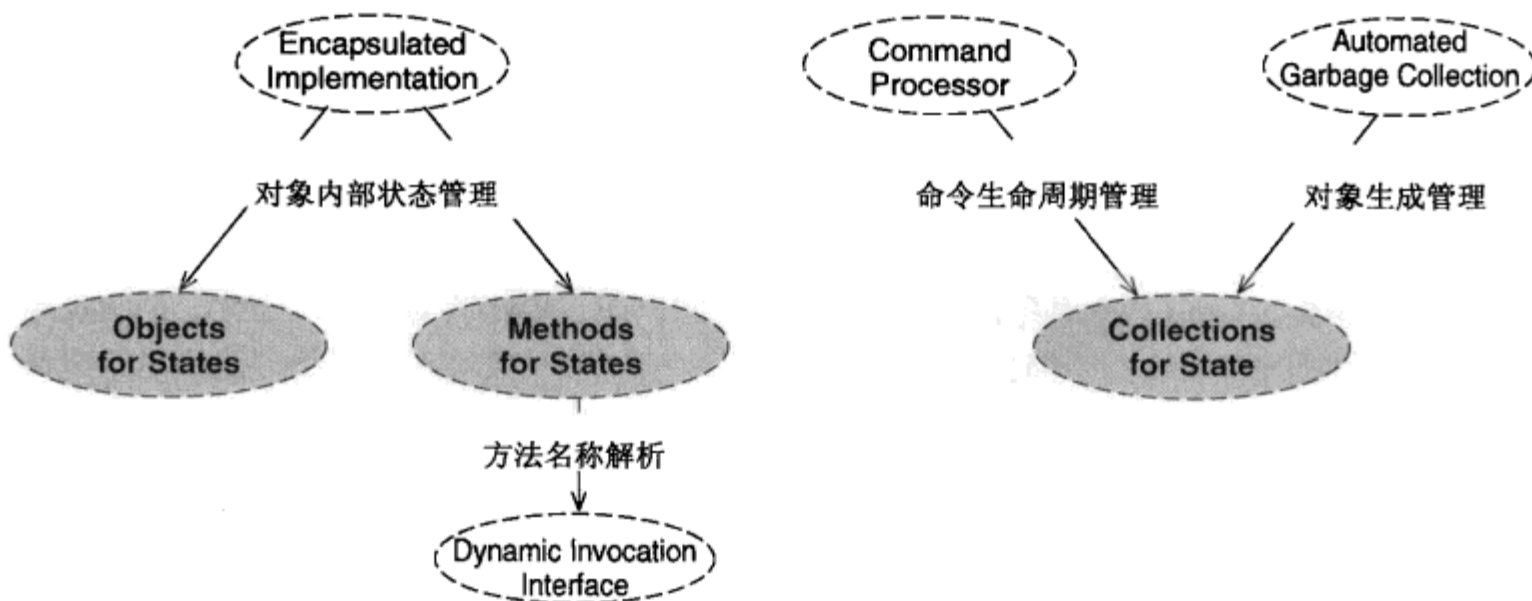


图 19-1

19.1 Objects for States*

在实现Encapsulated Implementation (181) 的时候……有时内部状态改变时其行为会发生显著变化，我们需要支持这种情形。



对象的行为可能是模态化的，其模式依赖于对象的当前状态。然而，将相应的多部分（Multi-part）条件代码硬性写入对象实现中会影响它的可理解性以及将来的开发。

例如，代表用户界面控制器（User Interface Controller）的对象要采用与当前视图（View）状态相适应的方式对通用事件做出响应。而视图可能根据选项和验证情况有多种模式。采用switch和if语句将状态机硬编码进去不能有效地扩展，并且仅适用于只有很少几个状态和受影响的方法的情况。状态机不同的方面不能独立改进，例如特定状态的行为、连接状态的转换逻辑以及新状态的集成等。

因此，将对象状态相关行为封装到状态类层次结构中，每一个不同的模态状态对应一个类。使用适当类的实例来处理对象状态相关的行为，转发方法调用（见图19-2）。

无论什么时候客户端调用模态对象的某个具有状态相关行为的方法，对象将方法执行分派给相应的状态类。模态对象创建后就和一个初始状态类的实例相关联，该状态类实现了初始状态的行为。当模态对象改变状态时，改变它所使用的状态对象，从而在新状态下保持正

确的行为。

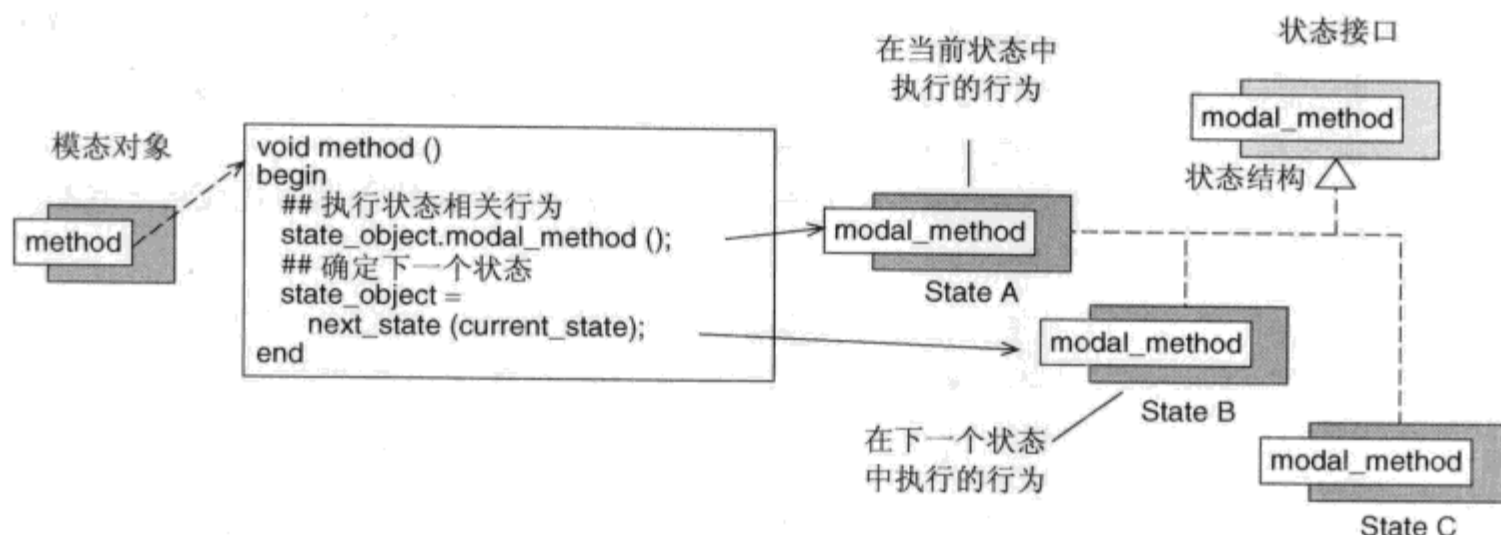


图 19-2



将状态相关的行为封装并组织在类体系中使得我们可以动态地为模态对象配置任意的状态类实例。扩展该类体系以便增加新状态或者修改现有的状态类都变得非常简单。

然而，Objects for States设计将责任分布在多个类中，这可能会使它在状态和与状态相关行为较少时导致过度的复杂。当状态的数量过多时，该设计也会变得难以管理，这是由于相应的类数目膨胀造成的。内嵌类或者将状态类置于几个包（package）或文件中有助于管理这种的复杂性。

状态类的实现通常是无状态的（别惊讶☺）。每个状态类接受其主对象或者其实例数据的引用作为每个方法的参数。这种状态无关性允许模态对象共享状态对象的实例，作为static数据被访问。然而，这种风格的编程所带来的间接性对于那些紧密耦合类的系统可能是没有必要的。在需要模式相关的状态时，将状态类做成状态相关的（stateful）可能更简单，例如基于事务的状态就是这种情况。

一般来说，切换对象使用的状态实例在实现上有两种选择。一种是在主对象自身内部实现该逻辑。另一种是由当前的状态对象来决定自己的“继任者”。每种选项都要确保在对象转换到新状态时正确地切换当前使用的状态对象。这时需要考虑的是到底要集中管理状态转换逻辑还是保留各种组合的灵活性。

19.2 Methods for States*

在实现Encapsulated Implementation (181) 的时候……有时内部状态改变时其行为会发生显著变化，我们需要支持这种情形。



对象的行为可能是模态化的，其模式依赖于内部的状态。然而，把这种模态化的行为硬性写入对象的方法中会给将来的开发带来麻烦。同时，将对象行为分派给对象群中的一个也会使得不同模式之间的协调和数据共享更加复杂。

例如，表示网络连接的对象在连接建立前、连接建立后和连接关闭后对方法调用的响应是不

一样的。

对象方法内的条件语句是表现这种行为的一种方式，但是状态机越复杂，对象的改进就越困难。将每个模式的行为封装到单独的对象中可以分解模态化的功能，但是在不同模式依赖相同数据或要求数据驱动的协调时会使得状态机过于复杂。

因此，将状态相关的行为实现为对象的内部方法，并使用数据结构以指向代表特定状态行为的方法（见图19-3）。

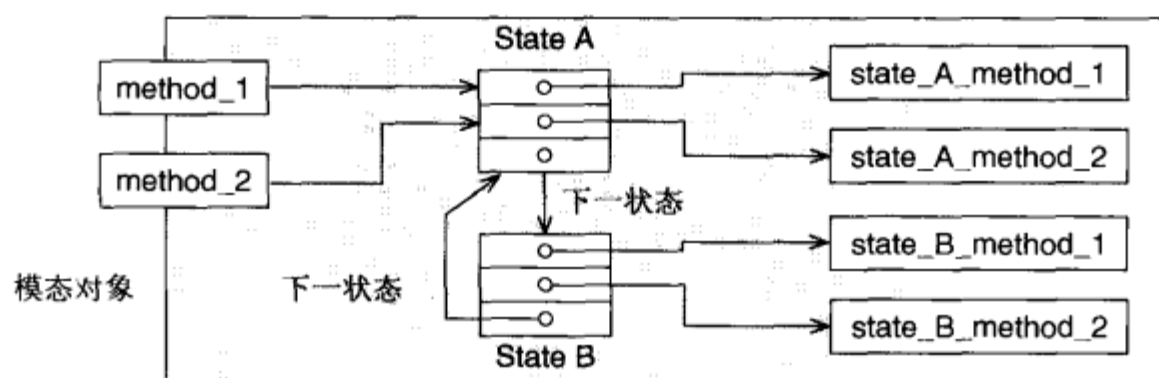


图 19-3

当客户端调用模态对象的具有状态相关行为的方法时，对象通过表示当前状态的数据结构将请求分派给内部方法引用。模态对象在创建时就应该与代表初始状态的数据结构相关联。当模态对象改变状态时，切换所使用的数据结构使其在新状态下执行正确的行为。



将状态相关行为封装在内部方法中允许模态对象在不同模式中共享数据和上下文信息，从而得到最好的性能和最小的资源使用。使用数据结构来指向状态相关的方法简化了对象状态机的配置和改进。

持有方法引用的数据结构可以有命名字段的类似于记录的数据结构，例如C++中的struct。也可以使用字典对象来放置私有方法的引用，这些私有方法分别对应于每个“历史相关（history-sensitive）”的公用方法。实际上，这种配置模仿了通常的多态方法查询机制，例如C++的vtable，并加入了一点定制、改进和智能性。当只有单一的公共方法与状态相关时，就不需要中间数据结构来表示模式，一个方法引用就够了。我们可以使用全局性、模块级或者类级变量保存每个模式所需数据结构或方法引用的单一实例。

所谓的方法引用可能是真正的方法引用，例如C++中的成员函数指针或者C#中的delegate，也可能是通过反射机得到的方法名，通过调用Dynamic Invocation Interface (167) 来执行。后者只在动态语言或者高可配置对象中其开销才是值得的，这时状态机可以在类的外部指定。

19.3 Collections for States*

当在Command Processor (199)、Automated Garbage Collection (307) 或者类似的集合管理设计中，管理服务请求对象时……我们经常需要根据对象的当前状态统一地处理其生命周期或者进行其他的操作。



对象的行为依赖于其当前状态时也许可以将其建模为单独的状态机。然而，有时候其客户端将这些对象的行为看作是模态的，而对象本身并不关心任何客户端所关注的状态模式。

例如，垃圾回收器会区分被引用的对象和未被引用的对象，但是对象自己并不关心这一点——也不应当关心。让对象意识到它们自己的状态并允许它们自己来管理状态会使得其实现和客户端使用它们的方式耦合得太紧密。无论什么时候客户端改变了它的状态模型，所有对象都会受到影响，这会使得对象的维护和改进更加困难。在不同客户端对对象的状态具有不同的视角时情况会变得更糟。在客户端以集合方式处理相同状态的对象会引入额外的资源和性能损失，比如所有对象等待垃圾回收器删除。

因此，在客户端内部，将感兴趣的不同状态表示为单独的指向该状态对象的集合（见图19-4）。

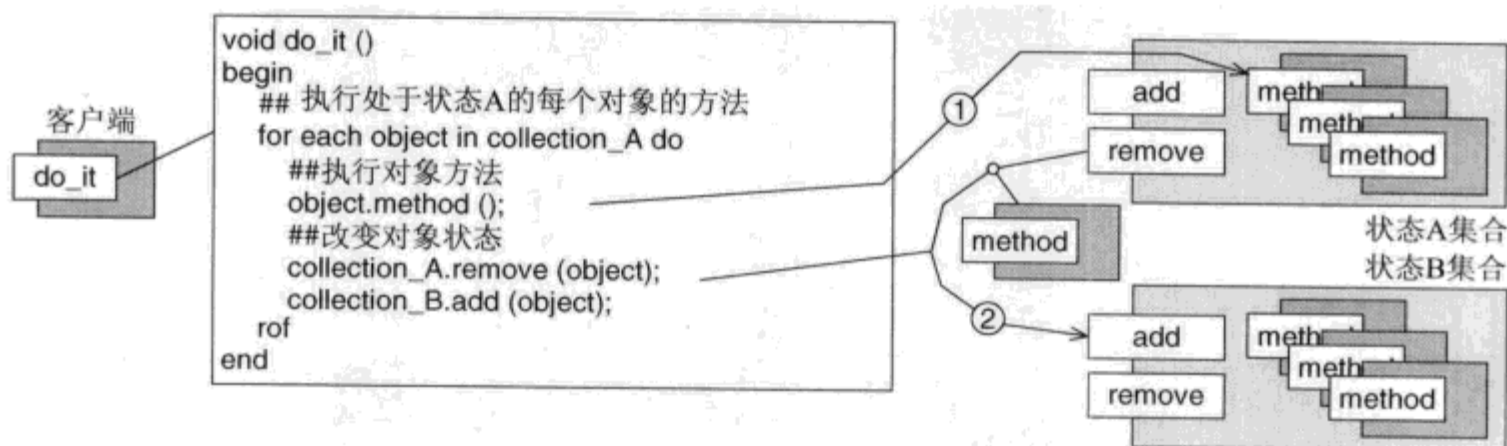


图 19-4

无论什么时候对象改变状态，就将它从代表源状态的集合移到代表目标状态的集合中。客户端只能调用某一个特定集合内对象的方法，这是由其所在集合所代表的状态决定的。



该对象所在的集合决定了对象的状态，所以不需要在对象内部表示其状态。为了优化选择处于特定状态对象的速度，除了这种基本的表现方式之外，我们还可以使用一些外在的表现方式。

特定状态的所有对象可以以集合的方式进行管理，这样对内存的使用较少，并且允许客户端把它们作为一个组来更有效地执行操作。客户端相关的状态模式可以在不影响对象类型的情况下实现，这可降低应用的结构复杂性，并支持对象和其类型的独立性和改进。

在每个客户端内部，至少对于每个感兴趣的状态要存在一个相应的集合：最简单的方案是互斥地表示每个状态。也可以提供包含其他集合的集合，例如除了包含特定状态对象的集合外，我们还可以提供一个包含所有对象的集合。

当状态数量增长时，Collections for States就不太适用了，因为集合——以及所有集合间的对象管理功能——会引入资源管理和性能开销。类似地，如果状态变化非常剧烈，由于在集合间传递对象的开销，Collections for States也不适用于这种情况。



阿姆斯特丹有轨电车车库
© Kevlin Henney

“资源”这个词涵盖了各种跟编程有关的对象和实体，包括数据库会话、同步原语、安全令牌、文件句柄、网络连接，甚至还包括分布式服务和组件等。资源范围涵盖从重量级的实体，如应用服务器组件进程，到细粒度的轻量级实体如内存缓冲区等。本章描述了21个模式，帮助管理提供给客户端的资源的生命周期和可用性，包括确保在需要时创建或获取资源，以及及时将它们删除或释放。

管理资源非常困难，而在分布式系统中有效管理资源就更困难了。应用的质量属性如性能、可伸缩性、灵活性、稳定性、可预测性、可靠性以及安全性，通常非常依赖合适的资源管理策略和机制。而要在各种需求之间寻求一种平衡就更麻烦了，因为要满足一个需求往往会影响到其他需求的满足。例如，灵活性通常会降低性能，而健壮性会因为使用检查点和恢复机制而降低可预测性。类似地，优化特定的用例，如最小化服务启动时间，会增加复杂性和常规用例的延时。

因此，为了有效地在分布式系统中实现应用的资源管理，必须应对以下挑战，从而在这些需求中争取达到适当的平衡。

- 性能。具有性能关键时间线（performance-critical timeline）的应用必须满足很多特性，包括低延时（动作和响应之间的最小时延），以及就单位时间内完成动作数目而言的高吞吐量。因为每个动作可能包含大量资源，尽量减少那些引入处理开销和延时的活动非常重要，这包括资源创建、初始化、获取、释放、销毁以及访问等。
- 可伸缩性。庞大而复杂的服务器应用通常需要多个资源来完成其功能。它们通常还有大量客户端多次访问其资源。在许多情况下，服务器是为特定使用情形设计的，如用户数目的期望峰值和平均值。这些情形可能会因为随着时间推移增加新需求而发生变化。例如，新的需求可能要求服务器处理两倍用户数目，以及十倍以上传输而不会影响系统性能。服务器应用的资源管理策略能满足这些需求才算是可伸缩的。
- 可靠性。可靠的服务应当一致并且不中断地满足客户请求。为了实现可靠性，应用必须仔细管理其资源。例如，如果一项事务和多个资源相关，则其最终状态必须是一致和持久的。此外，当性能或可伸缩性优化必须不能降低可靠性。
- 灵活性。应用中的一个通用需求是易于配置，这就要求应用属性可以在编译时、初始化时或者运行时被选择。灵活性高的应用为用户提供了这些自由。因此资源管理的机制也必须是灵活的，同时满足性能、可靠性和可伸缩型要求。
- 升级。长期运行的应用可能会在其生命周期内不断改进。理想情况下这些改进应当平滑可靠地进行，即使是在引入新资源的情况下也不应该产生太大影响。将整个应用关闭以完成升级通常是不可接受的，特别是对那些对可用性有严格要求的应用来说更是如此。
- 透明生命周期控制。资源客户端通常希望能够随时使用资源所提供的服务，而不关心资源生命周期管理的细节，比如什么时候以及怎样创建资源或者销毁资源，或者是否临时禁用或者清出资源。相反，有效的资源管理可能要求禁用那些很少使用的开销大的资源以便获取新的资源。很难使得资源生命周期控制既对客户透明，又能支持高效的资源提供环境。

因为上述挑战经常交织在一起，很难在处理某一方面的时候不影响其他方面。这使得资源管理更加复杂，并且促使我们使用经过时间检验的模式来应对这些挑战。我们的分布式计算模式语言中的以下21个模式在实现高效资源管理的同时提供了平衡相互冲突的挑战，帮助我们满足分布式并发性应用的需要。

- Container（容器）模式 (288) [VSW02] 为组件提供了运行时环境，以及组件正确执行所需的基础设施服务。
- Component Configurator（组件配置器）模式 (289) [POSA2] 允许应用在运行时加载或卸载组件实现而不需要修改、重新编译或者静态重新链接应用。它还支持将组件重新配置到不同的应用进程中，而不需要关闭并重新启动已运行的进程。
- Object Manager（对象管理器）模式 (291) [POSA3] 将对象使用和对象管理分开，从而能够显式地、集中地、高效地处理组件、对象和资源。
- Lookup（查找）模式 (292) [POSA3] 有助于查找和获取分布式对象和服务的初始引用。
- Virtual Proxy（虚拟代理）模式 (294) [GoF95] 在需要的时候才载入或创建开销较大的组件，

并可能在使用后从内存中删除。

- ❑ Lifecycle Callback（生命周期回调）模式 (295) [VSW02]提供对组件生命周期的明确控制。
- ❑ Task Coordinator（任务协调者）模式 (296) [POSA3]通过协调完成包含多个参与者的任务，维护系统的一致性，以确保要么参与者所做的所有工作成功完成，要么都不完成，从而确保系统状态的一致性。
- ❑ Resource Pool（资源池）模式 (298) [POSA3]避免获取开销很大的资源，并通过回收不再使用的资源来释放它们。
- ❑ Resource Cache（资源缓存）模式 (299) [POSA3]在资源使用完后不立即释放资源，以避免重新获取资源的昂贵开销。相反，它把资源保留在内存中并重复使用而不需要重新创建。
- ❑ Lazy Acquisition（延迟获取）模式 (300) [POSA3]在系统执行中将资源获取操作推迟到尽可能晚的时刻以优化资源使用。
- ❑ Eager Acquisition（提前获取）模式 (301) [POSA3]在资源实际使用之前获取和初始化资源，使得运行时资源的获取可以预测而且快捷。
- ❑ Partial Acquisition（局部获取）模式 (303) [POSA3]通过将资源获取分解为多个阶段来优化资源管理。每个阶段根据系统限制——如内存或其他资源的可用性等，来获取部分资源。
- ❑ Activator（激励器）模式 (304) [StSc05]自动根据可伸缩的需求激活或者停止由大量客户访问的服务，从而避免不必要的资源消耗。
- ❑ Evictor（逐出器）模式 (305) [HV99][POSA3]规定怎样以及何时释放资源，如内存和文件句柄，来优化资源使用。
- ❑ Leasing（租借）模式 (306) [POSA3]通过规定资源使用者如何在预定时间内从资源的提供者获得资源访问，来简化资源管理。
- ❑ Automated Garbage Collection（自动垃圾回收）模式 (307) 提供安全简单的机制来回收不再需要的对象所占用的内存。
- ❑ Counting Handle（计数句柄）模式 (309) [Hen01b]通过引入句柄对象作为共享对象的引用以简化共享对象的生命周期管理，并用它来跟踪共享对象的引用数目。
- ❑ Abstract Factory（抽象工厂）模式 (311) [GoF95]提供了创建和删除一系列相关或相互依赖组件对象的接口，从而避免客户端和具体类的耦合。
- ❑ Builder（建造者）模式 (312) [GoF95]将复杂组件的构造和析构从其表现中分离出来，从而同样的构造和析构过程能创建和删除不同的表现。
- ❑ Factory Method（工厂方法）模式 (313) [GoF95]通过提供创建组件的方法将组件创建的具体细节封装起来，而不是让客户自己实例化具体的类。
- ❑ Disposal Method（销毁方法）模式 (314) [Hen02b]通过提供销毁对象的方法将组件销毁的细节封装起来，而不是让客户自己销毁对象，或者由垃圾回收器来完成。

上面简要介绍的模式可以划分成7组，每一组解决特定的资源管理问题。

第一组模式，Object Manager、Container和Component Configurator规定整个资源管理框架。由于每个模式有不同的动机，它们经常被联合使用。

- Object Manager将对象和资源的使用与其生命周期管理和访问控制相分离。
- Container为组件对象提供完整生命支持系统，为组件提供生命周期管理、基础设施服务以及资源等，包括安全性、事务和持久化。Container通常是基于一个或多个Object Manager之上的。
- Component Configurator通过允许在应用的整个生命周期中——甚至在执行时——替换和重新部署组件实现，来补充上述两个模式。

图20-1和图20-2描绘了这3种模式和我们分布式计算模式语言的集成。

通过Object Manager我们为有文献记录的3种模式提供了一个不同的理解角度，它们是Manager[Som97]、Object Lifetime Manager[LGS00]以及Resource Lifecycle Manager [POSA3]。所有这3种模式解决了组件和资源管理的重要方面，如获取资源的访问和控制其生命周期。然而没有一个模式全面深入地覆盖其相应的主题，它们也没有阐述高效组件和资源管理包含的所有方面。Object Manager不仅包含和扩展了Manager、Object Lifetime Manager和Resource Lifecycle Manager的关注焦点，还阐述了组件和资源管理的其他方面。例如，它包含了资源共享以及从内存中暂时移除资源。此外，Object Manager描绘了其功能的许多实现选项。

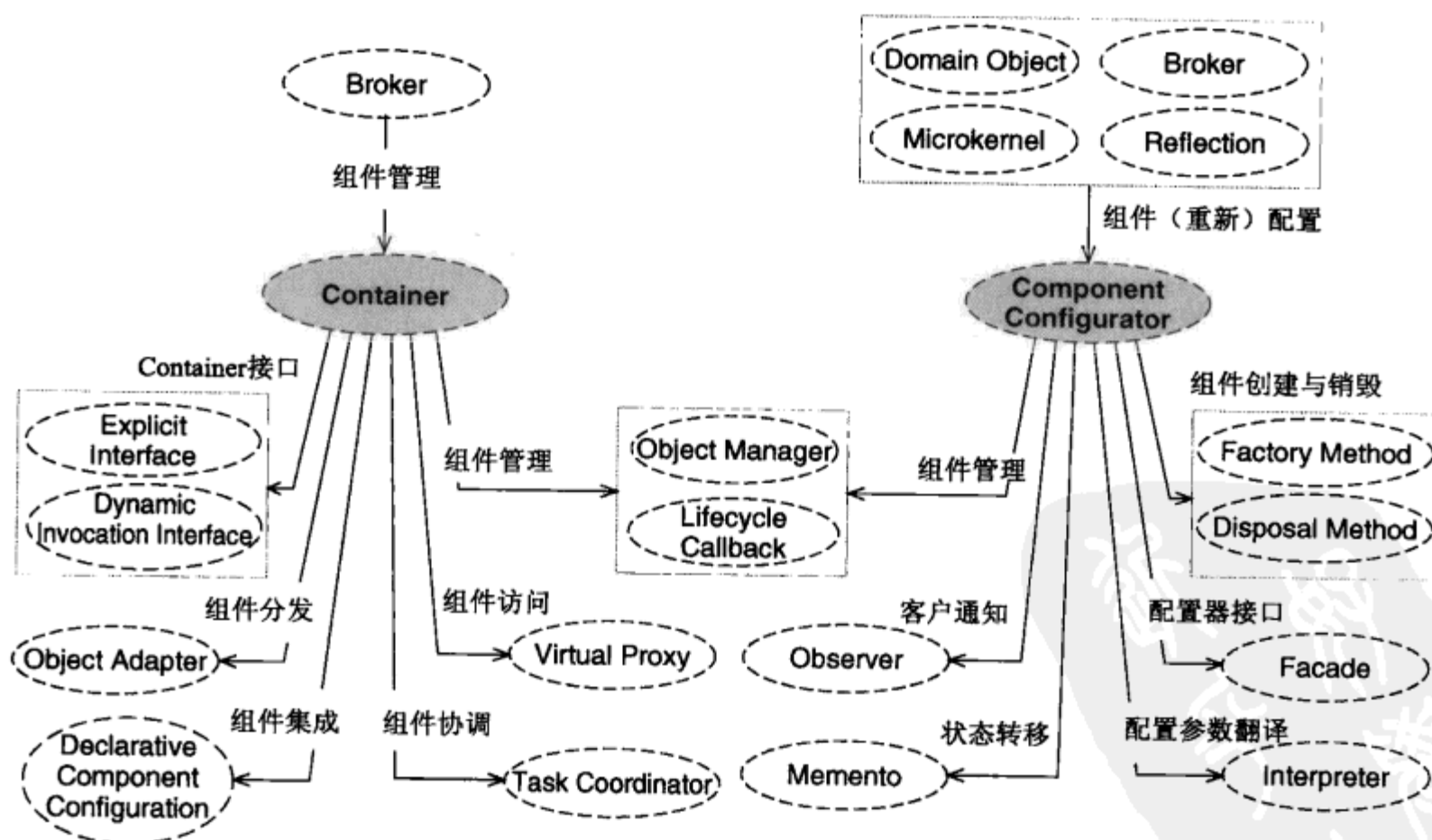


图 20-1

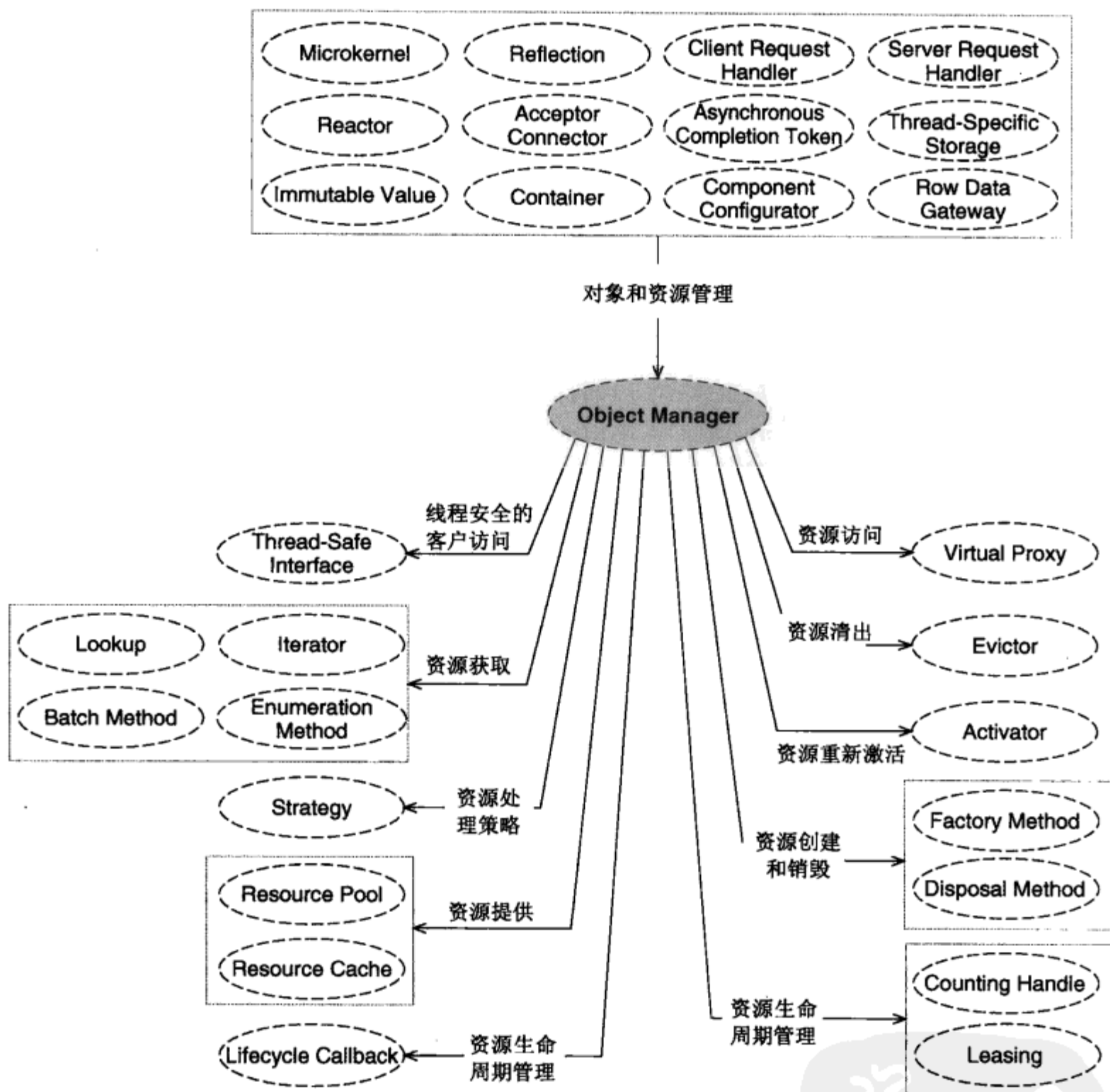


图 20-2

下一组模式针对有效资源管理的不同实现方面。

- Lookup有助于查找和获取具体的资源。
- Virtual Proxy可以帮助我们对客户隐藏所有资源管理活动。使得客户以为资源一直存在并可用，即使在它暂时失效或被删除。
- Lifecycle Callback定义了资源管理相关的对象和资源应当支持的接口，从而可以显式控制对象的生命周期。

- Task Coordinator支持会修改组件状态的任务在多个分布式组件对象和资源中的协调执行。我们在原来的 *Patterns for Resource Management* [POSA3] 的 Coordinator 模式前增加了“Task”前缀，以表明其具体的应用范畴是协调任务，而不仅仅是协调事务。
- Resource Pool和Resource Cache通过保留一套“现有的”资源降低了对开销较大的资源获取和释放的需求。这两个模式的关键不同在于Resource Pool不保留所管理资源的标识——这些资源被认为是相同的，也就是说，客户端得到一份资源，而不是某个特定的资源。相反，Resource Cache保留所管理资源的标识，当缓存并不包含所请求特定资源时则请求失败，即使存在其他属性完全相同的资源。因此Resource Pool有助于优化对无状态资源的访问，例如内存、线程，或者无状态的应用服务，而Resource Cache可以优化对包含状态资源的访问。
- Resource Pool和Resource Cache与 *Patterns for Resource Management* [POSA3] 中的 Pooling 和 Caching 模式相对应。然而，因为POSA4中所有模式名都是以名词短语命名，所以我们决定重新命名这两个模式。新的名字还反映了这两个模式在我们分布式计算模式语言中的特定角色。

上述6个模式针对的都是有效资源管理的最基本问题，它们在先前描述的Object Manager、Container和Component Configurator资源管理框架的绝大多数实现中都有使用。图20-3和图20-4显示了上述模式是怎样和我们模式语言联系起来的。

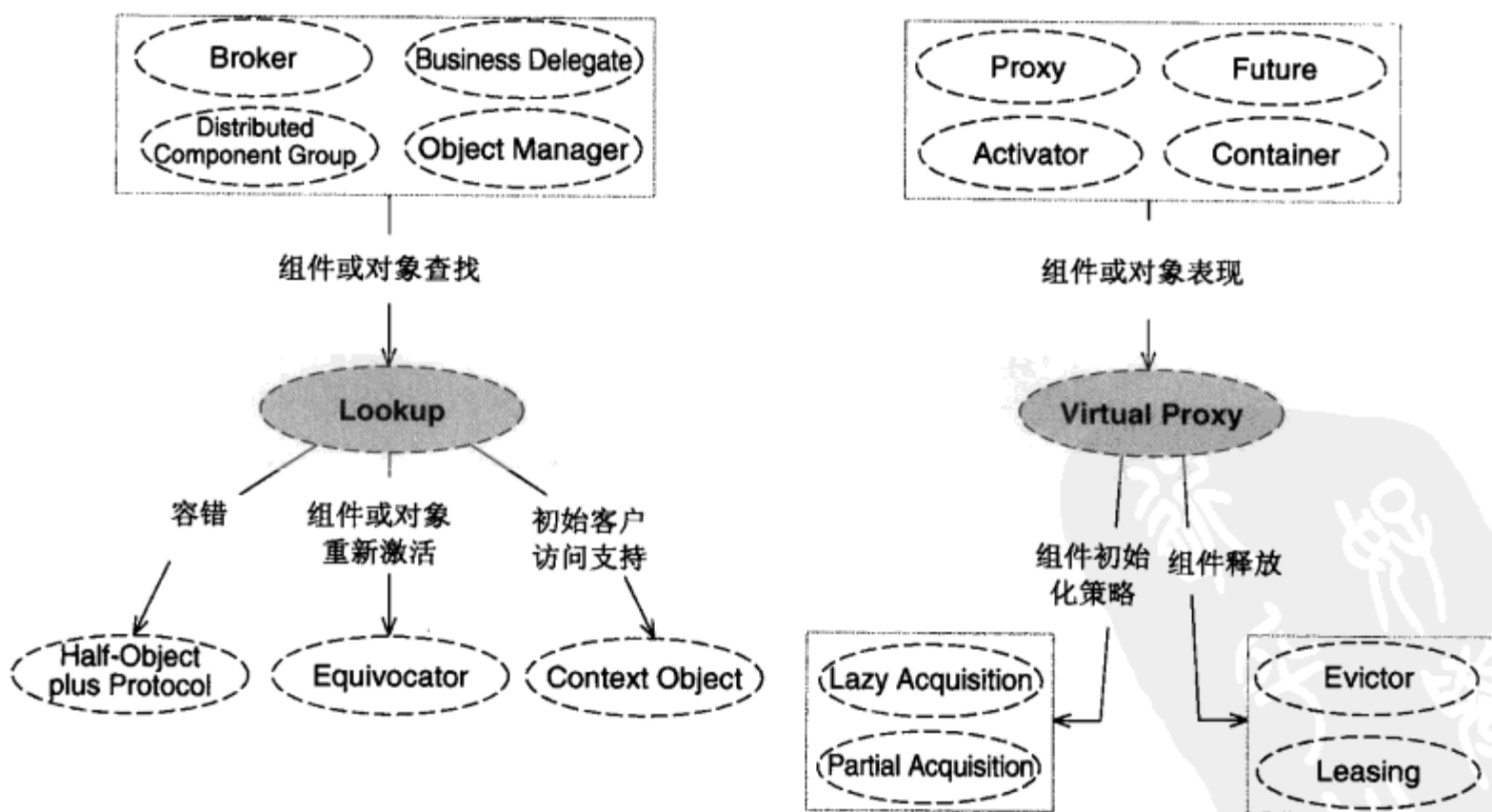


图 20-3

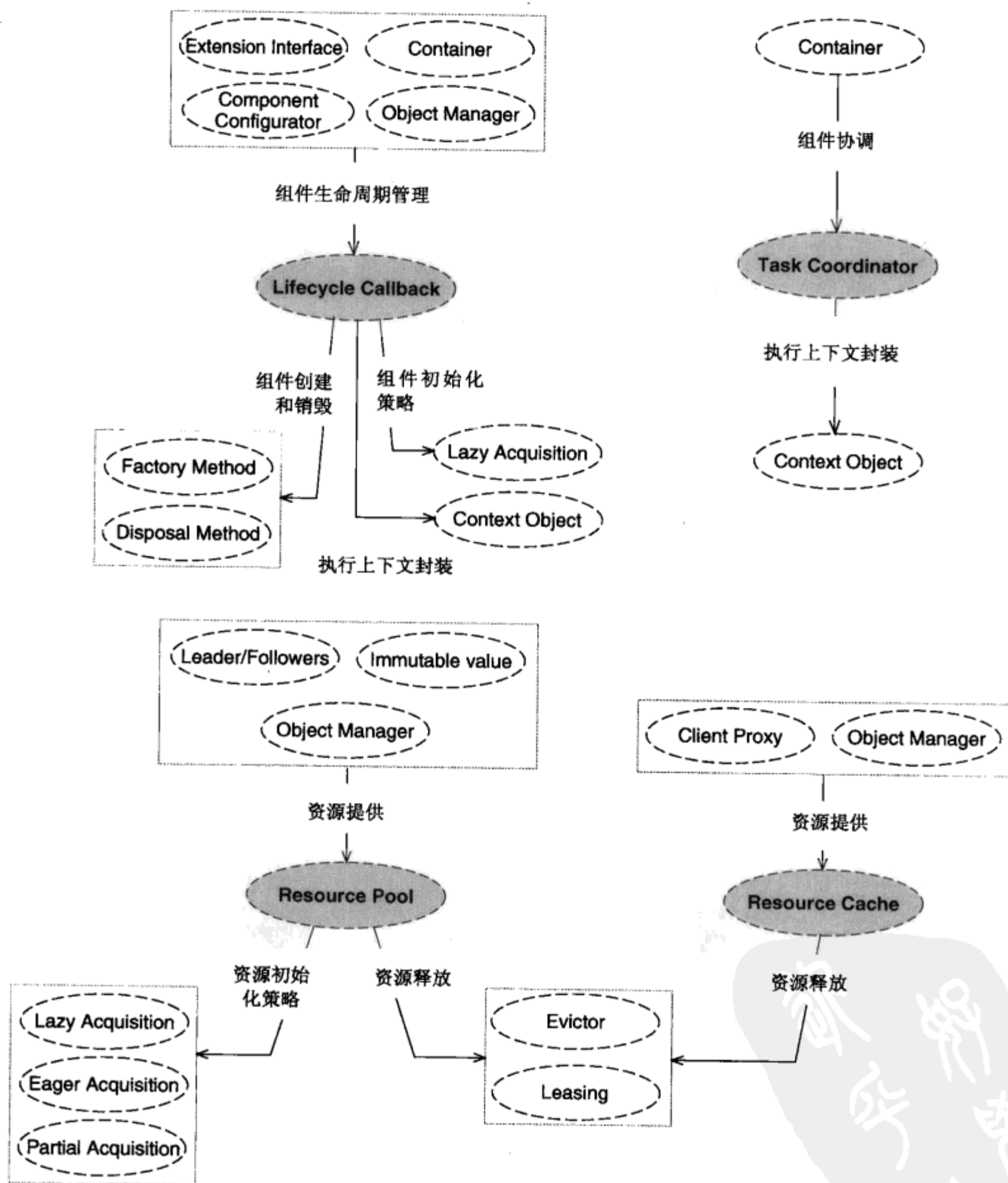


图 20-4

第三组模式针对了3种不同的资源获取策略：Eager Acquisition提供了提前获取策略；Lazy Acquisition提供了按需获取策略；Partial Acquisition综合了这两种策略，其资源的核心部分提前获取，而其他部分按需获取。考虑选择哪种特定获取策略是对性能和可用性与资源消耗(如内存占用等)进行折中。尽管Eager Acquisition确保资源在需要时是可用的，但是该方案会增加内存占用，当只是偶尔使用或者根本不使用资源时是无法接受的。Lazy Acquisition确保内存只在特定资源实际使用时产生开销，但是会在第一次使用时引入性能和可用性的损失。Partial Acquisition尽力来平衡折中，一方面考虑性能和可用性，另一方面考虑内存占用。

图20-5描绘了Lazy Acquisition、Eager Acquisition和Partial Acquisition彼此之间的关系以及和我们分布式计算模式语言中其他模式的关系。

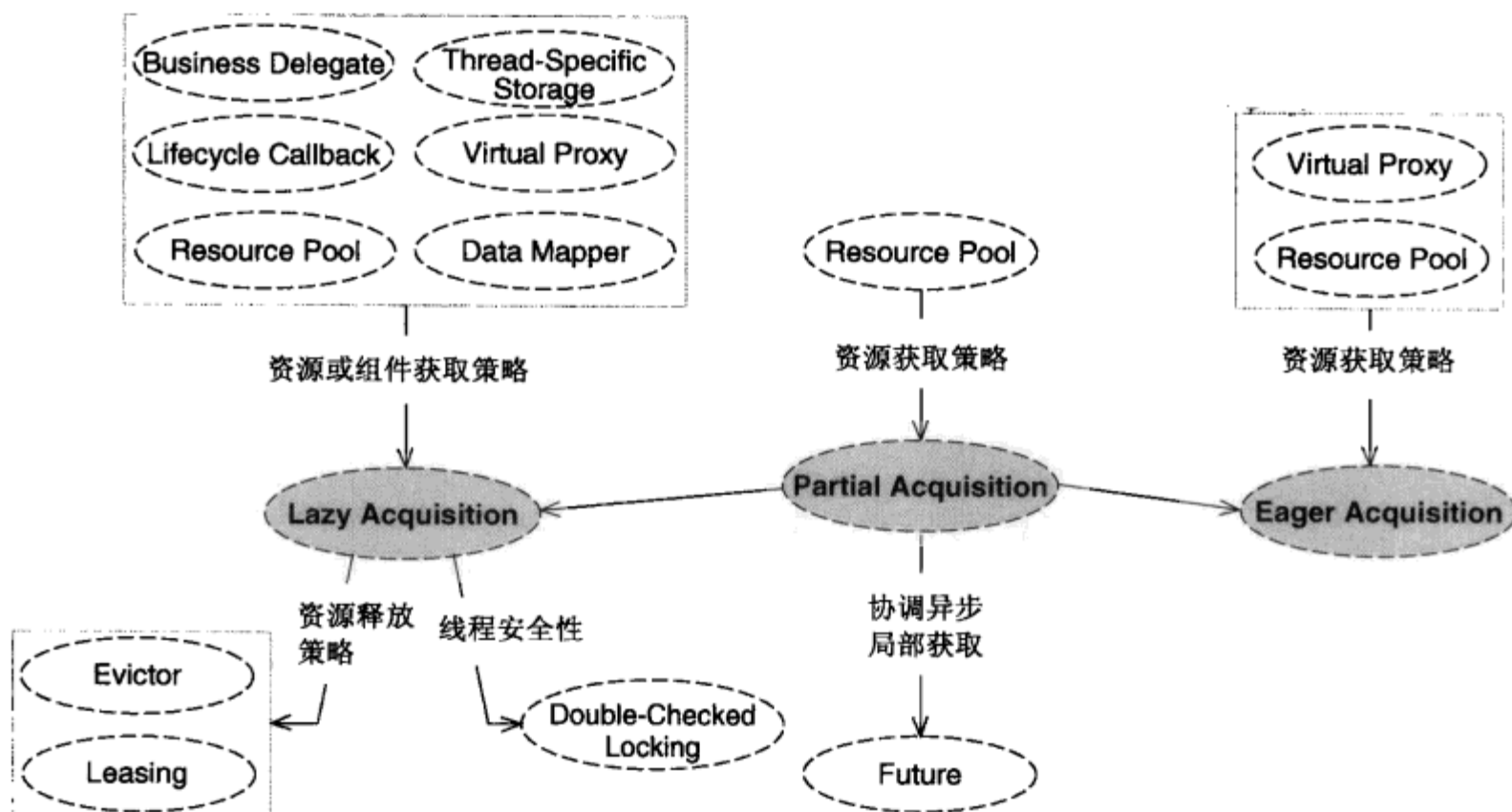


图 20-5

Evictor和Activator是一对相互补充的模式，以支持集中式的（重新）激活和释放资源。Evictor可以在某个时刻及时地全面释放资源，例如释放最近没有使用的缓存资源。然而，如果资源仍然能被其他客户引用到，例如容器中的组件对象，则Activator可以在下次客户访问时重新激活它们。

因此，Activator可以被视为Lazy Acquisition的具体例子。它将资源的获取延迟到系统生命周期晚期，例如在安装或运行时。尽管这两个模式很相似，但它们在抽象层次解决不同的问题背景。Lazy Acquisition定义了分配资源的广泛策略，从共享的被动实体——如内存或连接，到主动实体——如服务。Activator与之相反，是一个更专注的模式，解决资源受限的分布式计算环境中服务激活和失效问题。不过，Activator也有和Lazy Acquisition一样的各种优缺点，比如性能的

不确定性。

图20-6展示了Activator和Evictor是怎样集成进我们模式语言中的。

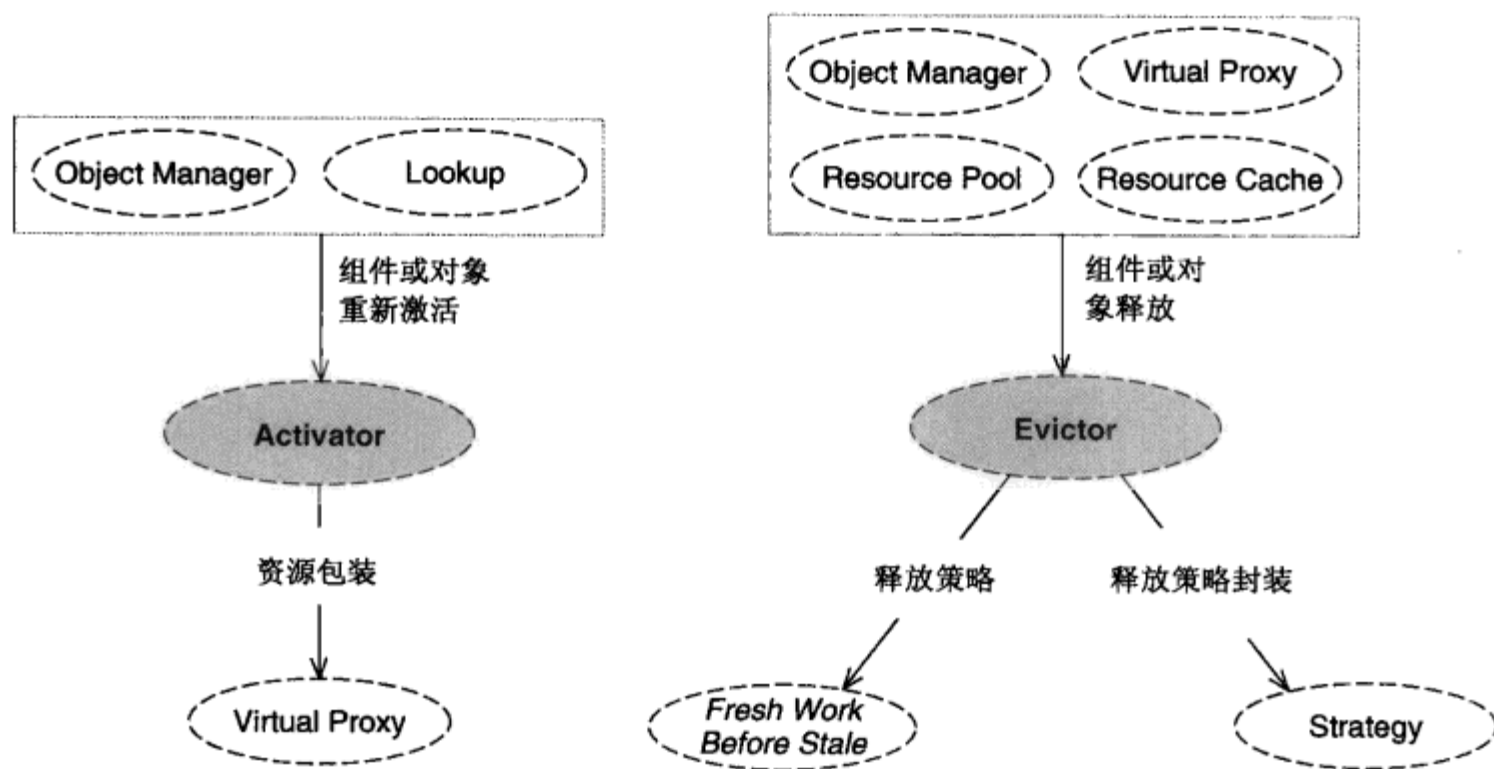


图 20-6

下一组的3个模式针对多个客户共享的对象和资源的释放策略。

- Leasing规定了资源可用的预定义时间。当所有授权租约到期后，资源可以由资源管理环境安全收回。
- Automated Garbage Collection为资源释放定义了“获取并忘掉”的策略。垃圾回收器定期监控应用中的所有活动资源，并释放不再被客户引用的资源。
- Counting Handle实现引用计数。只要还有客户引用该共享对象或资源，它就不能由客户发起释放。只有最后一个使用资源的客户发起销毁请求时资源才被释放。

在这3种模式中选择哪一个要考虑的因素包括以下方面。

- 谁发起释放资源的请求？其选择包括资源本身，例如Leasing，资源的使用者，例如Counting Handle，或者其他东西，例如Automated Garbage Collection。
- 什么时候释放资源？其选择包括在没有客户使用时，例如Counting Handle，或者在稍后某一时间，例如Leasing和Automated Garbage Collection。

所有3种模式的共性是资源释放对客户端来说是透明的——它们不需要关心这一点。

需要注意的是我们用Reference Accounting[Hen01b]中的Counting Handle模式替换了A System of Patterns[POSA1]中的Counted Pointer名称。这是基于下面两个原因而做出的。

- Counting Handle比Counted Pointer更准确，而且是丰富的模式语言的切入点。
- 该替换允许我们比用Counted Pointer更深入和一般地来描述引用计数。

图20-7描绘了上述3个模式是怎样连接到我们分布式计算模式语言中的。

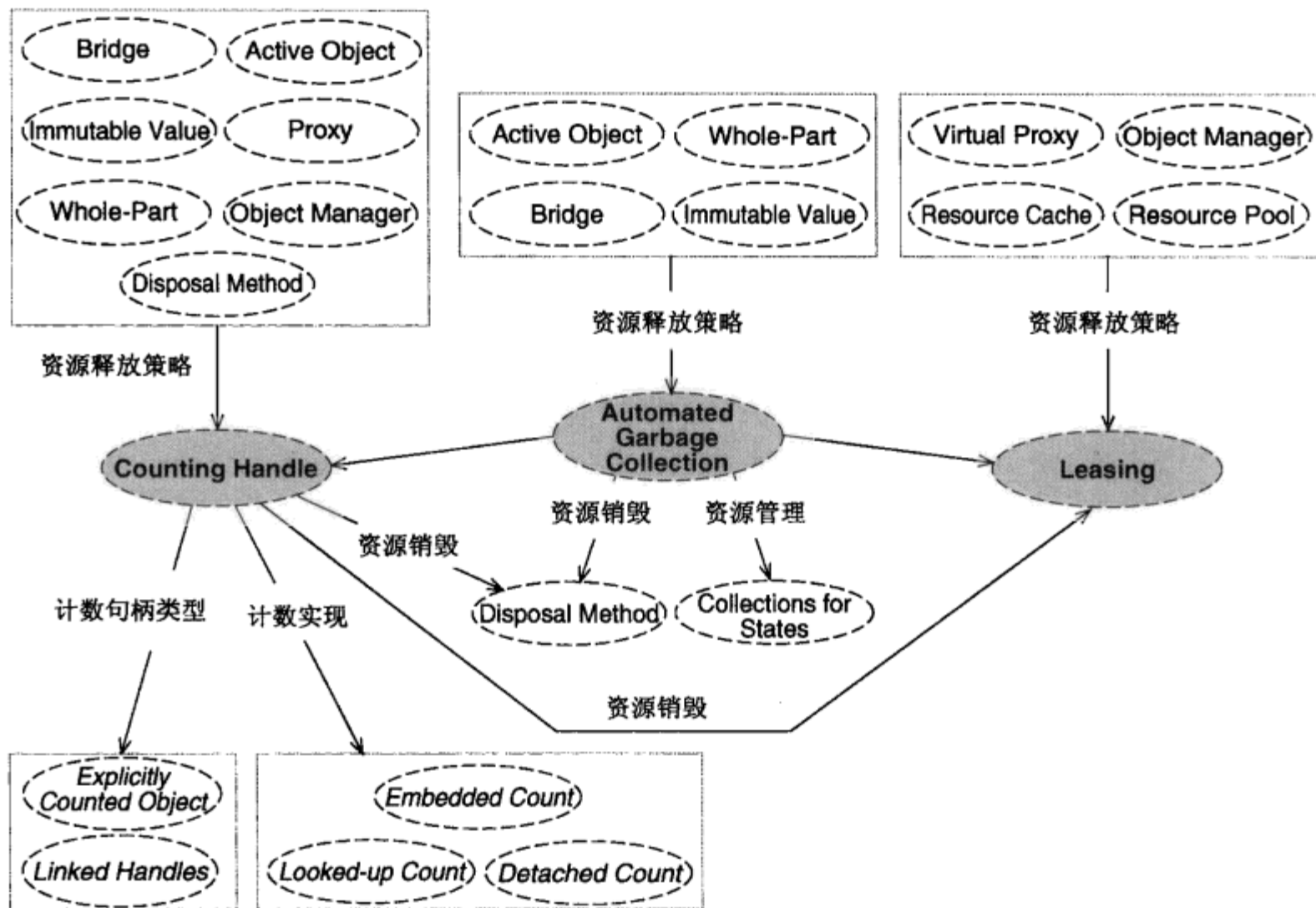


图 20-7

本章中的最后4个模式针对对象创建和析构的不同方面。

- Abstract Factory支持相关对象集的一致创建和销毁。
- Builder针对由多个部分构成的复杂对象的灵活创建和销毁。
- Factory Method和Disposal Method是两个互相补充的模式，通过简单并易于使用的接口隐藏了对象创建和销毁的细节。

图20-8展现了上述4个模式在我们分布式计算模式语言中是如何使用的。

熟悉Gang-of-Four著作的读者可能注意到本章略过了两个对象创建型模式：Singleton和Prototype[GoF95]。没有包括Singleton是因为它引入的问题——如删除、线程安全性以及配置性等——比它解决的问题还要多。至于Prototype，尽管该模式在其它领域中有应用——例如用户界面框架等，但我们的模式语言中并没有用到。

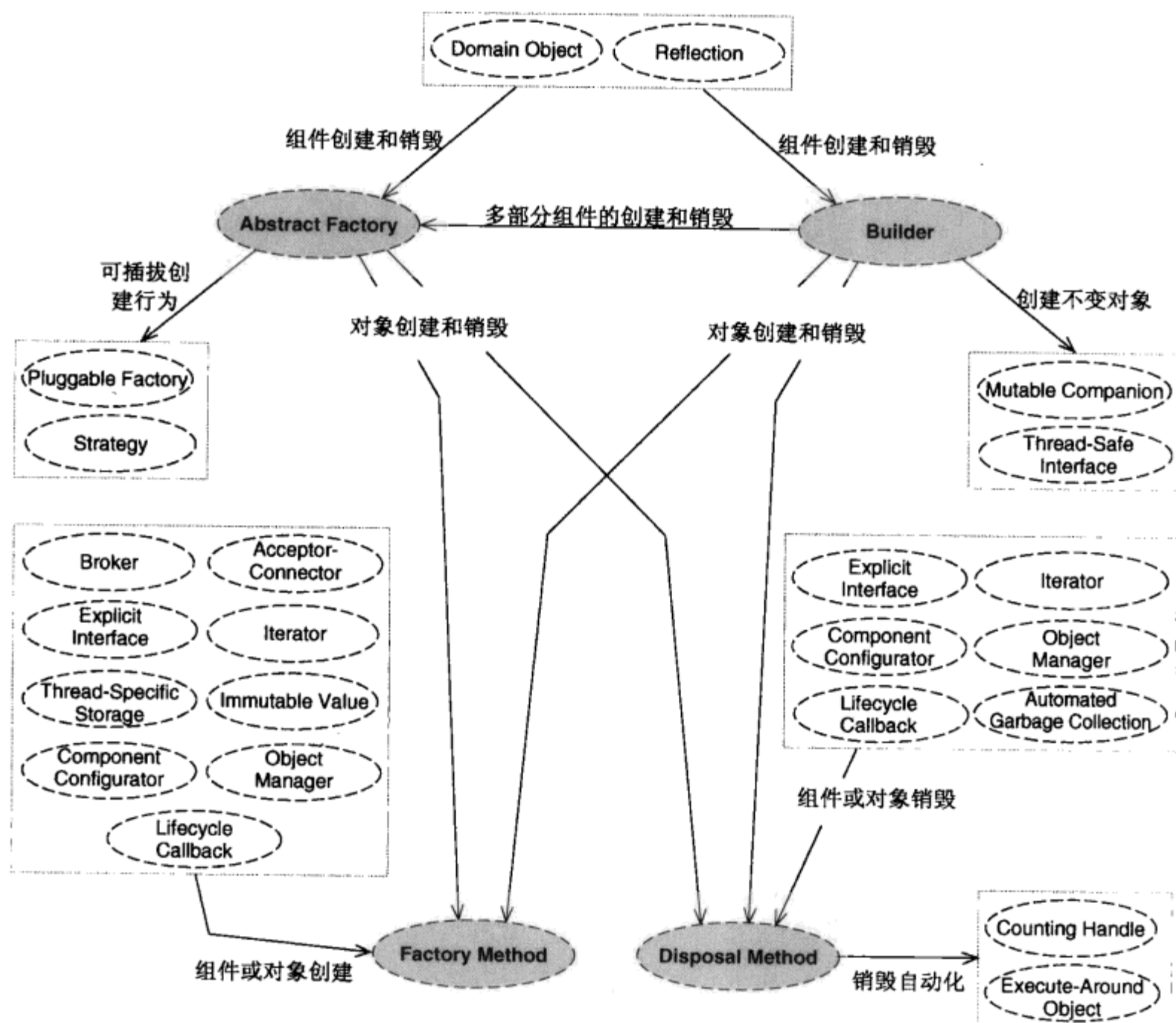


图 20-8

20.1 Container*

当为基于组件的系统实现Broker (137) 或者在分布式系统内使用组件时……我们通常会试图让组件和其所处环境的技术细节相分离。



组件实现了用于组成应用的业务逻辑或者基础框架逻辑，并且能够自成一体。然而，由于组件可能会部署在大量不同的应用和平台上，不能假定它们会有特定的执行场景和技术环境。

有些应用将组件作为事务的一部分，而另一些则不是。类似地，有些应用使用组件时要求满足严格的安全策略，而另一些则不要求。组件还可能被用在不同的系统平台中，这些平台访问系统资源的方式不同，例如持久化和并发性机制。让组件直接处理这些问题使得它们和平台耦合紧密并且增加了实现的复杂性。应当能够做到将组件集成到不同的应用部署场景并在各种系统平台

中执行，而不需要程序员介入。

因此，定义一个 Container 来为组件提供执行环境，并提供必要的技术框架支持以便将组件集成到特定应用使用场景和特定系统平台中，而不需要将组件和应用或平台紧密耦合在一起。

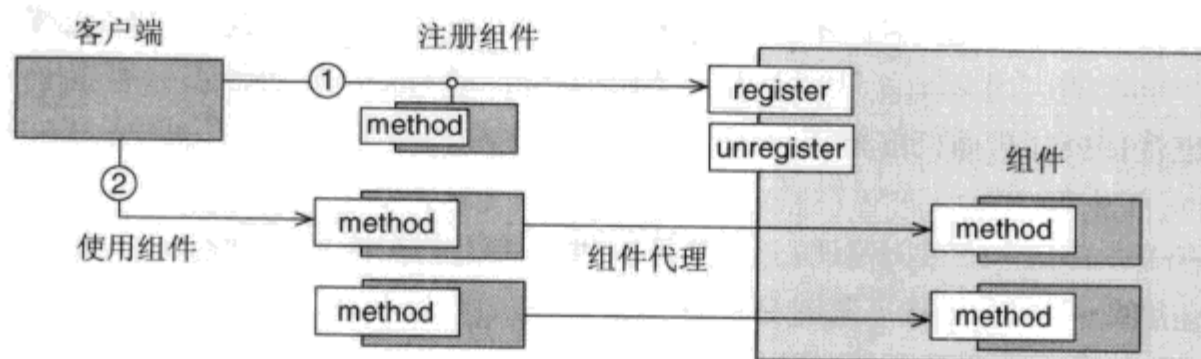


图 20-9

使用 Container 来初始化组件并为所管理的组件提供运行时环境。定义一些操作使得组件可以访问到其他组件端口的连接，以及访问通用的中间件服务，例如持久化、事件通知、事务、复制（replication）、负载均衡以及安全等。提供一种方式将组件按照声明规范自动集成到 Container 中，而不必用代码编程实现。

Container 将组件使用与将组件集成到特定应用或平台环境分离开。Container 允许开发人员专注于提供组件和应用的核心逻辑，而不是手动处理环境因素。本质上 Container 为组件提供了“生命支持系统”。

组件可以通过注册自己的方式集成到 Container 中，使用 Declarative Component Configuration (270) 指明需要使用的框架服务和资源以及它希望怎么使用它们。Container 解释这些配置项并自动完成具体的组件集成。为了从 Container 的生命支持系统中受益，组件可以通过 Explicit Interface (163) 直接访问其 Container，或者通过 Lifecycle Callback (295) 得到服务细节的通知。

Container 起到 Object Manager (291) 的作用，管理注册的组件。类似地，Container 使用一个或多个对象管理器来管理为组件提供的中间件服务和资源。Virtual Proxy (294) 实例为客户端提供了组件对象一直存活的假象。为了访问不提供这种代理的组件对象，Container 可以提供 Dynamic Invocation Interface (167)。

Task Coordinator (296) 有助于控制在多个组件对象上进行的会导致组件状态改变的任务，以确保所有参与的组件对象成功完成，或者全部都不执行。Container 使用 Object Adapter (256) 将从 Dynamic Invocation Interface 或底层中间件接收到的组件调用从通用格式转换成被调用组件的特定接口方法。

20.2 Component Configurator*

在实现 Domain Object (121)、Broker (137) 中间件，或者 Reflection (114) 和 Microkernel (113) 架构的时候……我们需要在运行时支持灵活的组件配置。



贸然将不够成熟的应用提交到组件实现的特定集合，可能不灵活并且缺少远见。有些决定必须等到生命周期的后期才能做出，甚至是在部署之后，而强迫应用负担不需要使用的组件，或者不能利用更好或更新的组件都是我们所不期望的。

组件有自己的生命周期，它们不断改进和成熟。例如，新版本提供更好的算法或者修正缺陷。因此使用这些组件的应用应当能从中受益。类似地，依赖特定软件或硬件环境的组件实现必须在环境改变时进行替换。然而，需要高可用性的应用是无法忍受宕机的，因此必须降低升级对运行系统的影响。

因此，将组件接口和其实现解耦合，并且提供一种机制在应用中动态（重新）配置组件，而不需要关闭应用或者重新启动（见图 20-10）。

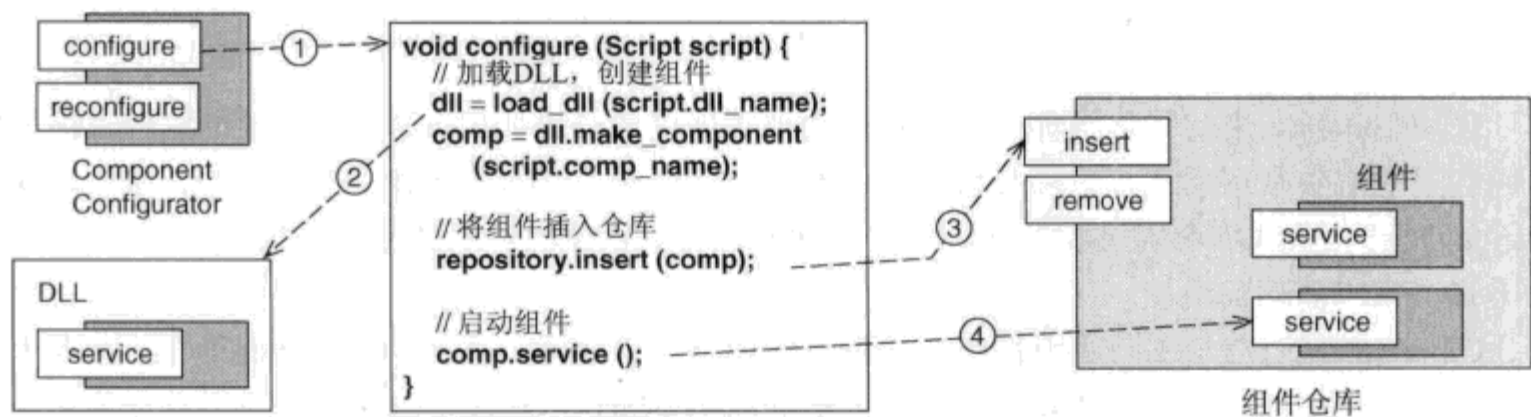


图 20-10

将组件组织为适当的部署单元从而可以动态加载，并提供框架来支持在运行系统的明确控制下（重新）配置组件。通过组件仓库集中管理受配置的组件，并提供API或使用某种形式的脚本在运行时（重新）配置指定的组件集合。



Component Configurator通过允许在应用的整个生命周期——甚至在执行过程中——替换和重新部署组件实现，增强了灵活性。类似地，与将不需要的组件静态连接到应用中不同，应用只需要为它们实际使用的组件付出时间和空间上的开销。

为了支持有效的组件（重新）配置，所有组件应当定义一个公共的管理性的Lifecycle Callback (295) 接口，包括初始化组件并启动执行、关闭组件并清空资源，暂停和恢复组件执行，以及访问组件当前执行状态信息等操作。Component Configurator框架使用该接口来（重新）配置每个组件。这个管理性接口可能还提供一个协议，比如Observer (237)，当组件终止时通知客户，以及为新版本传递状态。组件的状态以及和其他组件的关系可以通过Memento (242) 传递给新版本，在替换过程中则由组件仓库缓存。将组件组织成可以动态加载和卸载的动态链接库（DLL），并为每个DLL提供Factory Method (313) 来创建组件对象，以及提供Disposal Method (314) 在不再使用对象时销毁它们。

Component Configurator内部包含一个组件仓库，作为Object Manager (291) 来管理所配置的组件。Component Configurator的接口通常是一个Facade (171)，对客户隐藏其内部结构并将请求

分配给适当的参与者。它可能还包含一个Interpreter (258) 以处理配置指令，如果这是些简单的脚本语言指令的话。

20.3 Object Manager**

在实现 Client Request Handler (143)、Server Request Handler (144)、Reactor (150)、Acceptor-Connector (154)、Asynchronous Completion Token (155)、Microkernel (113)、Reflection (114)、Thread-Specific Storage (228) Immutable Value (231)、Container (288)、Component Configurator (289) 或者 Row Data Gateway (321) 设计的时候……我们经常需要管理某些特定类型对象的访问控制和生命周期，以及对象的资源和对象之间的关系。



在应用中有些对象——比如资源或者服务器端的组件对象——其访问控制和生命周期管理过程中都必须分外小心，否则就会影响到维护和使用效率，甚至出现错误。然而，如果要在这些对象内部实现这些功能则会增加这些对象的复杂性，并增加使用和改进的难度。

同样，客户端也不应该负责这些具体对象的管理工作，因为那样会将客户端与对象的具体类型、访问约束和生命周期策略耦合起来。这种方式也会使得对象发现变得更加困难，比如很难实现根据一个键值找到对应的对象。最终，这些依赖会增加应用内部的耦合度和复杂度。理想情况下，客户端应该仅依赖于对象的应用接口，而不应该依赖于对象的其他细节。

因此，将对象的使用同对象的生命周期和访问控制分离开来。引入一个单独的 Object Manager，负责管理和维护这些对象（见图 20-11）。

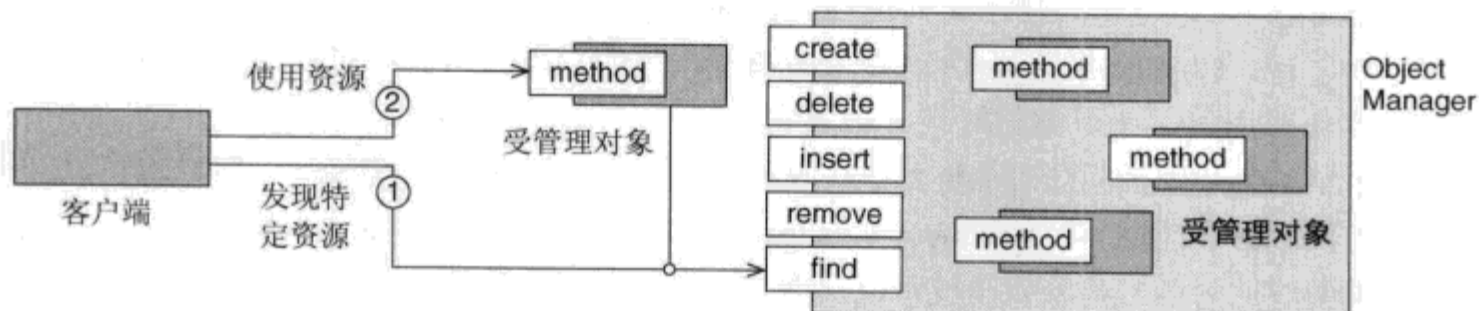


图 20-11

客户端可以使用Object Manager来访问具有某种职能的对象。如果请求的对象不存在，Object Manager可以根据需要进行创建。客户端也可以通过Object Manager显式地请求创建对象。某些情况下，客户端可能已经创建了对象，它也可以将对象移交给Object Manager进行管理。Object Manager还负责控制对象的销毁，该动作可以是对客户端透明的，也可以根据客户端的请求进行销毁。



Object Manager使得受管理对象和客户端都不必关心对象的生命周期管理和访问控制方面的细节。它将特定类型对象管理方面的细节集中在一个经过良好定义和封装的对象类型中，而且我们可以方便地找到该对象类型。

在有些应用中对于每一种类型的受管理对象只提供了一个Object Manager。例如，一个Object Manager负责处理线程，另一个负责处理连接。我们也可以在应用中根据目的和背景的不同提供多个Object Manager。比如为符合某一套管理策略的一个对象组提供一个Object Manager。如果Object Manager由多个线程共享，我们需要为它提供一个Thread-Safe Interface (224)。当然，如果对象只会使用在自己的创建线程中使用，则给每个线程提供一个Object Manager可能是更简单、更高效的。

客户端使用Object Manager提供的查找服务来请求访问对象。Lookup (292) 服务允许客户端根据对象的名字、属性或者其他键值来查找特定的对象。Iterator (173)、Enumeration Method (174) 或者Batch Method (175) 支持在多个对象间遍历，而无需暴露Object Manager的内部结构。

Object Manager有多种方式可以维护受管理对象。如果部署分布广泛、变化多样，则我们可以使用Strategy (266) 对象或者类型来实现策略的参数化。而对于有些情况，一个简单的设置方法接口就足够了。

Resource Pool (298) 可以用来保持固定数量的同类型、同ID对象始终有效，它往往用于管理重要的、持续使用的计算资源，比如进程、线程和连接。与之相反，Resource Cache (299) 则用来管理只存在一段时间的那些对象。为了避免影响到应用的服务质量，缓存可以销毁不再使用的对象并释放相关的资源，以便供其他对象使用。Evictor (305) 支持以受控的方式从缓存中移除不常用的对象。清出的对象仍然可以由客户端引用，并且可以通过Activator (304) 重新激活，以便供客户端使用。当然如果客户端通过Virtual Proxy (294) 来访问对象，重新激活的工作则由代理(proxy)来实现。

为了防止过早地释放活跃对象，Object Manager可以使用各种对象移除策略。Leasing (306) 使得Object Manager可以指定在哪个时间段内对象的引用是有效的，并且客户端可以更新其租期。在租约过期之后，Object Manager可以安全地销毁相应的对象。与其相对应的是Counting Handle (309)，仅当不再引用某个对象的时候才销毁它。

Object Manager中管理的对象要么是从内部创建，要么是由客户端提供。客户端可以利用注册功能将外部创建的对象移交给Object Manager，尽管我们可以使用Factory Method (313) 来实现对显式对象创建的封装。对象创建也可以对客户端是透明的。虽然由客户端来创建对象提供了某种灵活性，但同时也削弱了客户端的内聚性，增加了Object Manager实现资源管理方面优化的难度，同时也容易引入对象管理权方面的错误。

由Object Manager维护的对象必须在适当的时刻销毁。通过注销(deregistration)功能，客户端可以从Object Manager那里夺回管理对象的职能。Disposal Method (314) 可以显式地删除对象。在关闭应用的时候，Object Manager往往要求销毁所有受管理的对象，以确保正确地释放所有对象所使用的资源。

我们还可以让所有的对象都实现同一套Lifecycle Callback (295)，这样Object Manager就可以统一地控制它们的生命周期，包括创建、清出、重激活，以及最后的销毁。

20.4 Lookup**

在实现Broker (137)、Business Delegate (170)、Replicated Component Group (189) 或者Object

Manager (291) 的时候……我们经常需要查找并取得资源、对象和服务的引用，这些引用可能是本地的也可能是远程的。



在分布式系统中，一台服务器可能会为客户端提供多种服务。客户端在启动的时候不必知道服务器提供了哪些服务。同时，服务器上的服务也可能随着时间的推移而增加或者删减。如果客户端不知道哪些服务是有效的，它就无法使用这些服务。

在分布式系统中让客户端发现服务的一种方式是将服务的地址硬编码在客户端软件中。这种方法显然是非常不灵活的。理想情况下，服务器应该可以高效地、以可扩展的方式发布自己的服务——对客户端来说也应该能够方便地查找服务。然而，以广播的方式来实现这种发布从带宽和处理时间上讲都会引入太大的开销。

因此，提供一种 Lookup 服务，分布式系统中的服务在可用的时候将自己的引用进行注册，在不可用的时候则注销（见图 20-12）。

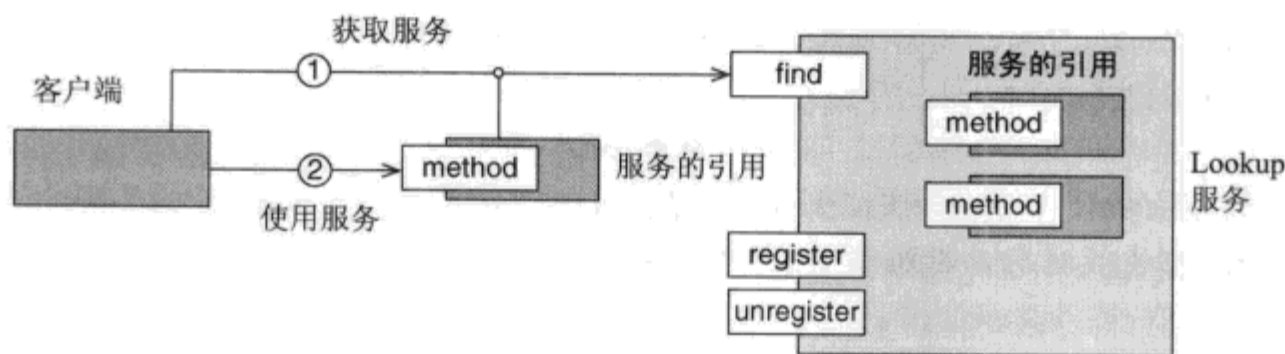


图 20-12

系统中客户端可以使用Lookup服务来获取已注册服务的引用。



Lookup对客户端和服务而言是一个“情报交换中心”，它允许客户端访问服务器上的服务，而无需让客户端将服务器或者服务器上服务的位置或者引用硬编码在客户端中。同样，服务器也不需要知道要访问其服务的客户端的位置。

组织Lookup服务有两种方式。

- 集中式。关于服务的信息保存在一个单独的地方。Lookup服务对这些信息进行持久化保存，即使系统发生故障也可以恢复正确的状态。这种实现方式比较直观，但是伸缩性较差，而且可能引入单点失败。
- 分布式。一组Lookup服务周期性地发布其注册服务的有效性。在这些Lookup服务中可以使用某种形式的组通信协议（group communication protocol）来完成相关信息在联盟服务之间的多播（multicast）。通常，分布式Lookup服务都适合用Half-Object plus Protocol (324)来实现。虽然分布式Lookup服务实现上更为困难，但是它的伸缩性较好，而且可以避免单点失败。

服务的引用可以与用来描述服务的一组属性和服务所提供的接口关联起来。Lookup服务在内部维护这些信息，使得客户端可以基于查询选择一个或者多个服务。我们可以在客户端需要定位

某个服务的时候使用Activator (304) 来（重新）启动某个Lookup服务，这样做的目的是减少分布式系统中活跃的Lookup服务的数量，减少资源消耗。

为了跟Lookup服务通信，客户端和服务端需要一个接触点（access point）。如果事先不知道该接触点的位置，客户端和服务端可以使用某种协议来找到它，这可能需要跟预先配置好的引导服务器（bootstrap server）进行交互，或者需要消息广播机制。可用的Lookup服务可以在响应消息中提供有关其接触点的信息。

客户端需要能够非常方便地访问Lookup服务，所以Lookup服务通常都是作为其初始化上下文的一部分，客户端在访问这些服务的时候或者是通过众所周知的名字或者是通过Context Object (243)。

20.5 Virtual Proxy**

在实现基于Proxy (169) 的接口、Future (223)、Object Manager (291)、Container (288) 或者Activator (304) 设计的时候，经常使用到某些开销甚大的对象……我们需要降低那些不必要的或者不常用的资源的获取成本。



有时候创建一个对象可能会消耗大量的内存和时间。而如果这些对象根本就不会使用，或者不会立即使用，这些成本就有些浪费了。当然，当我们需要访问这些对象的时候，它们必须是可用的。

一下子从一个巨大的数据库中把所有的对象（对应于数据库中的行）全部加载进来不仅会消耗大量资源，而且也是没有必要的，特别是我们往往只需要其中的一小部分对象。与之类似，使用一个Container来管理大量的服务器端对象，会消耗大量的内存和活动对象表的空间。如果这些对象使用得比较少，就很容易造成资源的浪费。理想情况下，对于不会用到，或者不会长期使用的资源，我们不当引入过多的开销。管理资源生命周期不当妨碍资源用户。

因此，为当前内存中尚不存在的对象引入一个代理（proxy）对象。这个代理对象可以处理一些简单的请求，比如查询目标对象的键值，但在需要完整的对象行为的时候就会创建并初始化目标对象（见图 20-13）。



图 20-13

代理对象与目标对象提供同样的接口。接口中的方法执行有两种方式，一种是根据在创建实际目标对象前预先保存的状态完成处理；另一种是根据需要首先创建目标对象，然后将调用传递

给目标对象。在后一种情况下，我们会首先检查目标对象是否存在，如果不存在则创建，然后将代理对象上方法的调用传递给目标对象。



Virtual Proxy为资源访问引入了一个间接层，使用结构上的偏移来支持时间上的偏移：目标对象的创建被推迟到第一次使用的时刻。由于间接层需要加入一个额外的对象，所以总是会引入一些开销，我们需要在开销和收益之间做一个权衡，同时还要看该应用在预计的负载下这种优化起作用的可能性有多大。

简单的Virtual Proxy通常是Lazy Acquisition (300) 搭配Activator (304) 实现的。对于有些对象来说，它们本身可以分为多个部分，这些部分可以分别获取而且加载开销各不相同，这时候我们可以采用Partial Acquisition (303) 来实现Virtual Proxy，这样的好处是可以将资源获取的开销分散得更为均匀。

使用Virtual Proxy，对象创建的开销并未消除，只是简单地推迟了。问题是，在真正初始化时出错的机会也同样被推迟了。所以除了要应对使用资源时可能发生的错误，客户端还要对付更为基本的错误^①，这有可能会影响到优化的透明性。

第一次访问时的开销被推迟到真正消耗资源的时刻。对于那些不常用的对象，一旦初始化完成，代理会一直持有其资源，即使以后不再使用或者极少使用。这就可能导致资源的浪费，甚至最终可能会使得资源枯竭。当然，我们也可以在第一次访问之后立即通过代理传递一个销毁命令，及时地删除该对象。然而，在某些情况下，这又会因为再次获取（释放）而引入更多的开销。通过使用Evictor (305) 或者Leasing (306)，我们可以稍后再释放相关的资源，以便如果紧接着有人会用到该对象的话不需要再次获取资源。

20.6 Lifecycle Callback**

在实现Extension Interface (165)、Object Manager (291)、Container (288) 或者Component Configurator (289) 等设计的时候，由于这些设计中都需要考虑框架中对象的生命周期管理……所以我们需要保证框架中对象能够对框架所发出的与生命周期相关的事件做出正确的响应。



有些对象的生命周期是非常简单的：客户端在使用之前将其创建起来，在使用期间该对象保持正常状态，直到客户端不再使用时将其销毁。然而，有的对象的生命周期就要复杂得多了，它们不仅受到其组件生存环境需求和事件的影响，还要受到额外的资源管理技术——比如池和钝化（passivation）——的约束。

与前一种对象（由客户端负责创建和销毁）不同，后一种类型的对象其生命周期通常是由框架来控制，并且与应用的策略和架构方面的要求有关。为了节省内存和其他资源，我们可以在对象的生命周期中间将其钝化，并在下次访问的时候再激活。而如何执行这些操作只有对象本身才知道，应用并不了解。但是我们还是需要给应用显式地控制这些对象生命周期的能力。

① 这里指对象创建的错误。——译者注

因此，以接口的回调函数的形式定义主要的生命周期事件，并让框架对象支持该接口。框架使用这些回调函数来显式地控制对象的生命周期（见图 20-14）。

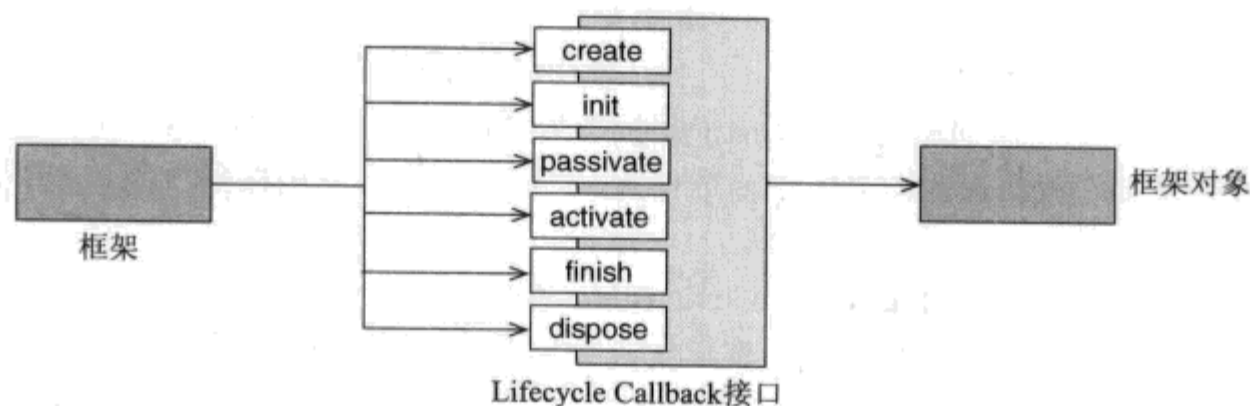


图 20-14

通常回调函数中包括了对对象的初始化与结束（finalizing）、钝化与激活、状态持久化与重新加载。



使用Lifecycle Callback框架便可以显式地控制组件的生命周期，而无需了解其内部结构，框架通过接口中适当的方法完成调用，而对象类型根据自己的结构和需要完成这些接口方法的实现。在一个框架中对象通常都是实现相同的Lifecycle Callback接口，目的是让框架可以统一地处理这些对象。

创建对象是一回事，而开始使用对象又是另一回事。因此，在Lifecycle Callback接口中，我们经常将这两个阶段明确地区分开：Factory Method (313) 负责对象创建；而Lazy Acquisition (300) 用于获取对象所使用的资源，它是由单独的初始化回调函数实现的。与两阶段创建相对应的是两阶段销毁：在接口中有专门的结束回调函数负责资源释放；还有一个Disposal Method (314) 负责真正的对象销毁。

如果对象需要访问组件环境的细节或者框架的服务，我们可以在每个回调函数中传入一个Context Object (243)。

生命周期回调函数常用于Object Manager (291) 和Container (288) 管理所属对象，包括组件对象的激活、钝化和移除。

20.7 Task Coordinator*

在实现Container (288) 的时候……我们需要确保，如果一个任务有多个参与者协作完成，其中只有一部分出现故障的话，不应当导致整个系统的状态出现不一致的状况。



在大规模的系统中计算机、网络 and 软件组件出现局部故障是很常见的问题。如果系统局部发生故障，有可能会应用出现状态不一致的情况，甚至导致系统瘫痪。如果相关的任务是分散在多个组件上，这个问题就更加严重了。

在很多应用中，一项任务通常都是由多个参与者共同完成的，比如资源提供者和资源消费者。每个参与者按照一定的顺序执行某一部分任务，只有所有的参与者都成功执行了任务，这项任务才能算是完全成功。对于一个成功的任务来说，它对系统所做的修改必须保证系统处于一致的状态。如果其中一个参与者出现了故障，而其他的参与者修改了应用的状态，而该参与者无法完成必要的修改。这样应用就可能无法产生正确的结果。

因此，引入一个 Coordinator，负责监控所有参与者对任务的处理和完成情况。Coordinator 确保，或者所有的参与者成功完成任务，或者如果其中一个出现了故障，系统要看起来就像该任务根本没有执行过一样，即返回到整个任务开始之前的状态。

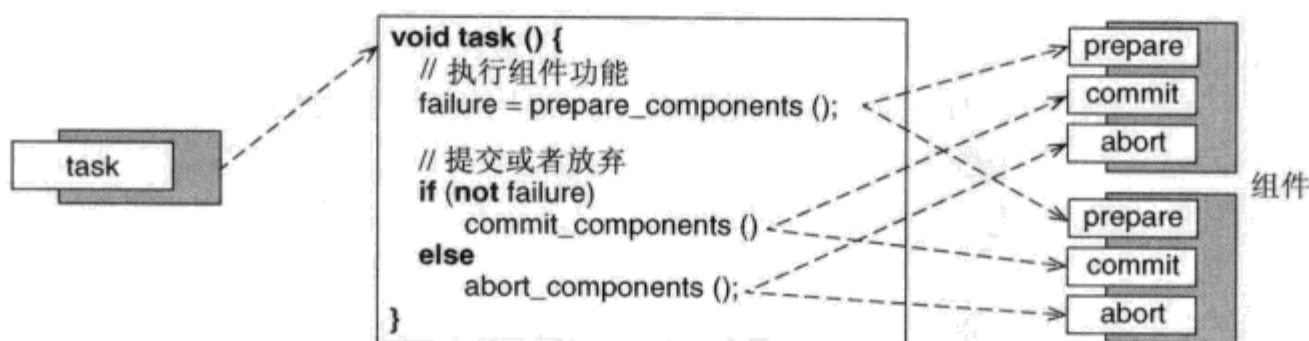


图 20-15

“两步走”模型是最简单的协调模型，该模型将通信量降到最低，而提高了分布和并发的机会，使得任务可以彼此隔离。在任务结束之后，不论是否成功，所有相关的资源自动释放。



Task Coordinator 确保一项由多个参与者共同完成的任务从客户端的角度来看是自动的。这种协调通过确保“多步状态切换”可控，并且可以安全地失败，保证了整个应用的一致性。

为了实现这种“两步走”的 Coordinator，将每个参与者的工作分为以下两步。

- 准备。在这个阶段，Coordinator 查询每个参与者，它所执行的部分任务有没有出现故障。如果参与者报告了潜在的故障，Coordinator 会停止整个任务的执行序列，要求所有已经成功执行了准备阶段的参与者放弃并回滚，返回到原始状态。既然所有参与者的修改都不是持久化的，系统的一致性自然可以满足。
- 提交。如果所有的参与者均通过了准备阶段，Coordinator 将告诉它们可以进入提交阶段，参与者可以执行实际的提交工作。既然每个参与者均可地成功地执行自己的任务，则提交阶段也应该可以成功，自然整个任务可以成功完成。

因此，任务的执行就变成了事务型的：任务看上去是原子的(atomic)，而系统的结果状态是一致的(consistent)。在事务处理过程中，状态修改是彼此隔离的(isolated)，成功的状态修改在整个事务执行过程中均是持久的(durable)。这被称为事务的“ACID”属性。由于在提交阶段仍然可能出错，有时候我们可以采用更为健壮的“三步走”提交的方式。

然而，尽管这种协调可以提供很好的伸缩性和完整性，但是 Coordinator 也引入了一些开销——比如任务分割和任务管理，以及在多个参与者之间传递事务性上下文(通常由 Context Object (243) 实现)。

20.8 Resource Pool**

在实现Leader/Followers (211) 并发模型、Immutable Values (231) 或者Object Manager (291) 等设计的时候, 由于经常出现资源周转的情况……所以对于某些无状态的资源, 我们需要能够快速地从获取和释放。



获取和释放资源——比如网络连接、线程或者内存——可能会引入一定的性能开销, 而每次获取和释放的开销有可能是不一样的。有些应用对性能和伸缩性要求较高, 这就需要能够高效地、以可预测的方式访问这些资源。

给定的资源访问策略必须是可伸缩的: 即使所使用的资源和资源用户增加了, 也不应当影响到资源获取的效率和稳定性。然而, 要保证性能的稳定, 对于同类型资源的获取和释放时间不应当变化太大。例如, 假设有一个服务器, 上面的每个请求都由一个独立的线程负责处理。对于那些频繁的短周期请求, 反复创建、准备和销毁每个线程的开销甚至会占到整个请求处理时间的一大部分。

因此, 将一定数量的有效资源作为一个 Resource Pool 保持在内存中。这样就不需要反复地从头创建资源, 从 Resource Pool 中取得资源的速度更快, 而且具有更高的可预测性。当应用不再使用某个资源的时候, 我们应当将其返回到 Resource Pool 中, 以备后面取用。

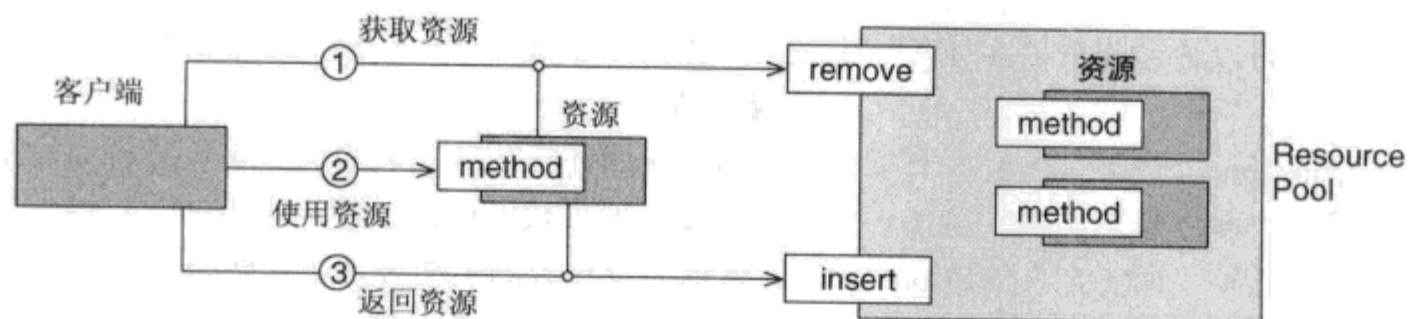


图 20-16

Resource Pool封装了资源获取、访问和管理的知识。要获取一份资源, 使用者必须知道其所在的Resource Pool。根据资源本身的特征不同和Resource Pool对资源的管理方式不同, 资源使用者可能需要(或者不需要)显式地将资源返回给Resource Pool。资源的释放可以随着Resource Pool的销毁而完成, 也可以通过显式的请求完成——如果Resource Pool支持这种接口的话。



Resource Pool使得我们不必每次收到请求都从头创建资源, 从而避免了性能方面的问题。通过将资源存储在一个池中, 资源的访问速度更快也更稳定。池中的所有资源具有相同的属性, 客户端可以平等地对待它们, 即客户端取得的是一份资源, 而不是某个特定的资源。

池中资源的数目可以是在创建时即确定的, 也可以根据某种策略动态增加, 比如固定递增或者指数递增, 这种增长可能是有界的也可能是无界的。有些Resource Pool可能会在不同的应用环境中使用, 增长策略也可以做成可配置的——这可以通过一套简单的设置策略参数的方法实现, 也可以通过Strategy模式实现。对于可扩展的池, 也应该支持显式的或者透明的收缩。如果Resource

Pool可以跟踪它所管理的资源,要对其进行调整并不困难。当然,我们可以使用更为复杂的方式,比如通过Evictor (305) 或者Leasing (306) 来管理资源清出。

池中的资源通常是在Object Manager初始化期间创建的,这时候我们可能会用到Eager Acquisition (301)、Partial Acquisition (303) 或者Lazy Acquisition (300)。Lazy Acquisition将对象创建推迟到第一次对其进行访问的时刻。相反Eager Acquisition则在访问之前将对象完整地创建起来,目的是让对象在创建之后立刻就可以使用了。如果创建会花费比较长的时间,我们可以使用Partial Acquisition来完成对象逐步的组装,这样可以减少第一步创建所花费的时间。

资源返回到池中之后,需要经过重新初始化才能供其他客户端使用。重新初始化确保资源处于“可用状态”,并满足安全方面的需求。

20.9 Resource Cache**

在实现Client Proxy (139) 或者Object Manager (291) 的时候……我们需要优化频繁访问同一套资源时的开销。



为几个有限的资源用户频繁地创建和释放资源会带来不必要的性能开销。对于某些应用来说,这样的开销会使得系统难以满足性能方面的需求,因此,我们需要降低常用资源的初始化和销毁方面的开销。

如果应用经常使用和销毁某种类型的资源——比如内存缓冲区或者线程,我们可以使用Resource Pool来提高系统的性能。虽然Resource Pool从整体上为应用提供了一个较好的解决方案,然而对于局部的情况——比如特定的组件或者子系统——可能就不适用了。所以,我们需要一个局部的、易实现的优化方法,而且这种方法的执行成本也不能太高。

因此,在使用完资源之后将其保存在内存缓冲区,而不是直接将其销毁。当再次需要同一资源的时候,从缓冲区提取该资源,这样就不需要重新创建了。

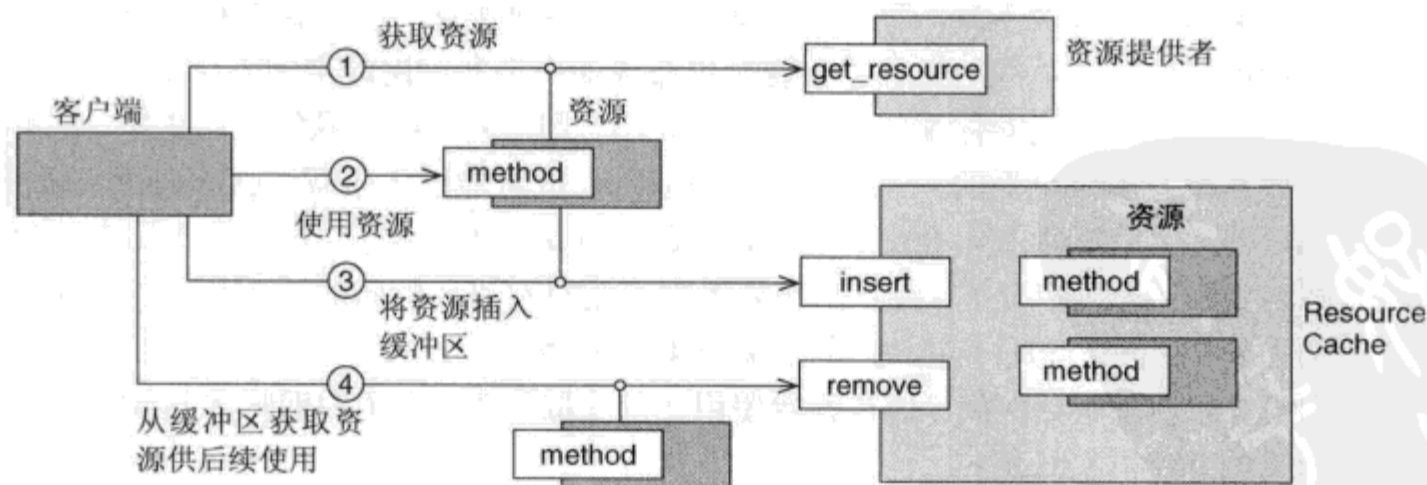


图 20-17

资源缓冲区临时缓存一份或者少数的资源,目的是为了可以快速取回这些资源。在获取某项特定资源的时候,资源使用者首先在缓存中查找。如果发现缓存中没有这项资源才去请求资源提

供者。有了这个缓存，我们就可以实现快速的本地化资源循环利用。如果缓存被销毁或者缓存已满，全部或者部分资源就需要释放。当然，如果缓存支持显式地请求释放资源的接口，我们也可通过该接口来释放资源。



通过将常用的资源保存起来而不是销毁掉，Resource Cache模式将（重新）获取和释放资源的成本降到了最低。理想情况下，资源只会创建一次——在第一次访问之前，或者在第一次访问的时候。同样，销毁也只有一次——在确定这些资源不再使用之后，或者应用结束的时候。

缓存中的资源即使具有相同的属性我们也认为它们是互不相同的。如果客户端请求的那份资源在缓存中没有找到，这个请求就失败了，哪怕缓存中有一份资源具有完全相同的属性。

我们可以使用Evictor (305) 来显式地清空缓存来释放资源，也可以通过Leasing (306) 隐式地完成该操作。如果应用还没有显式地要求释放资源就将其从内存中清出可能会引入一些开销。比如，在向缓存中插入新的资源的时候，如果发现剩余的空间不足以容纳新的资源，那么我们就需要销毁一些原有的缓存资源——通常我们是按照“最久未使用”或者“最不常用”的标准选择销毁哪些资源。当应用再次访问该资源的时候，我们就得从头创建或者激活它了。

缓存实际上是在空间和时间上的一个权衡，使用额外的空间提高系统的性能。缓存设计得越复杂，从开发的角度来看其维护难度就越高，而且一般来说它所带来的性能提高也越少。

20.10 Lazy Acquisition**

在实现Business Delegate (170)、Thread-Specific Storage (228)、Lifecycle Callback (295)、Virtual Proxy (294)、Resource Pool (298)、Partial Acquisition (303)、Data Mapper (320) 等设计的时候，由于初始化开销比较大或者不必在一开始就完成初始化……所以我们需要确保对象的创建和资源获取满足高吞吐量和可靠性的要求。



有些应用需要访问大量的资源，同时又要满足高可靠性的要求，我们需要找到一个方案，能够降低资源获取的初始成本，同时又保证在整个生命周期中资源使用的空间不会太大。

如果在系统或者子系统初始化期间获取所有的资源，则可能会导致系统启动变慢，这不仅没有必要，有时候甚至是不可接受的。而且，很多资源的获取可能过于乐观（over-optimistically），如果在应用的整个生命周期内都不会用到的话，这最初的获取就白费了。过度获取还会导致资源耗尽，妨碍资源重用。理想情况下，使用大量资源的应用只在必要的时候访问相关的资源，由于过早获取所引入的空间开销和早期启动方面的成本均应该避免。

因此，尽量推迟资源获取时间。仅在开始使用之前获取资源。在用户即将用到某个资源的时候才开始获取资源并将其交给用户。

延迟获取是一种乐观的优化方式，它将资源获取的成本产生推迟了——但并未消除。它的使用应该由资源提供者完全封装，使得资源使用者不需要关心相关策略和机制的细节。

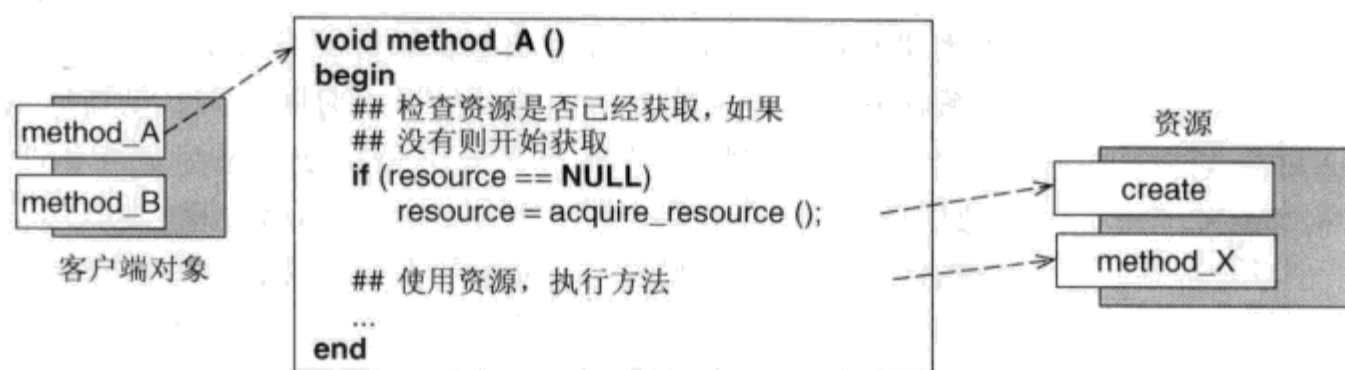


图 20-18



Lazy Acquisition奉行的是一种“明日复明日”的哲学：如果可以推到明天去做绝不今天开始。积极一点，也可以说是实践了精益原则的一种“推迟决断”，将实际的获取推迟到“尽可能晚”[PP03]。

Lazy Acquisition保证能够及时地获取资源，也就是说在需求出现，即将开始使用之前才获取。因此，在应用、组件和子系统生命周期中，Lazy Acquisition不会过早地引入资源获取方面的开销。进一步说，它不会为从来不会用到的资源浪费时间和空间。

然而，使用Lazy Acquisition通常会增加少量的空间开销。这是因为我们需要引入一些中间对象——比如Virtual Proxy，同时我们还需要一些空间来持有额外状态或者属性——它们用来标识获取的状态或者在获取资源时会用到。

获取方面的开销被转移了，但并没有消失，所以如果使用了Lazy Acquisition，系统在第一次用到某个对象的时候会相对慢一些。这会影响到系统的可预测性。类似的风险还包括在被推迟的资源获取过程中出现失败。所以我们没有办法保证在需要资源的时候它一定是可用的。

由于实际的获取对客户端是透明的，所以何时释放资源就不那么直观了，当然还包括是否释放，以及如何释放也有类似的问题。Evictor (305) 和Leasing (306) 是在后台释放资源的两种方案。

在并发应用中，Lazy Acquisition通常使用Double Checked Locking (225) 来保证同一份资源不会被并行的多个线程多次获取。把它实现为线程安全的设计是一种较为简单的锁方案，但是这种方案的开销比较大，因为不仅是第一次，每次访问资源的时候都会引入锁开销，这本来是不必要的。

20.11 Eager Acquisition**

在实现Partial Acquisition (303)、Resource Pool (298) 或者其他类似设计时，由于其初始化可能会引入较大的开销……所以我们需要保证对象的创建和资源获取满足高稳定性和高性能的需求。



每个应用都需要访问某种资源——比如内存、线程、网络连接和文件句柄。如果每次都在需要的时候才获取这些资源，对于那些对稳定性和性能要求严格的应用来说则可能无法承担这样的开销。

资源获取操作可能会引入大量开销，在执行任务时动态获取资源对于某些应用来说是不可行的。一般情况下，我们无法预测获取某个特定资源所需要的准确时间，尤其是在通用操作环境中。这可能会造成应用无法满足其稳定性的要求。在分层系统中，某一层中的原生资源 (primitive resources) 由下一层进行封装，如果在某一层实现了 Lazy Acquisition，那么对于其相对较高的一层获取资源时所花费的时间就更不易预测了。如果还要处理资源耗尽的情况，则任务实现就变得更为复杂。从运营的角度看资源耗尽更不可接受，所以我们需要尽可能地减少资源耗尽的情况。

因此，在使用之前尽早获取资源。保证在资源使用者请求之时资源必定是可用的。

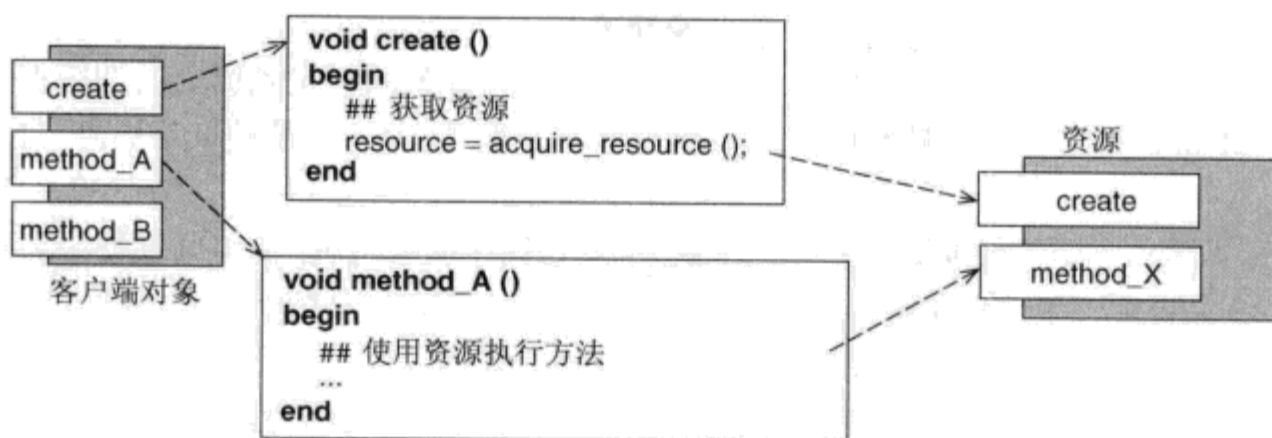


图 20-19

资源获取的时刻依赖于多个因素，包括创建相关资源所需的时间、什么时候会用到这些资源、创建资源的数量，以及它所依赖的其他资源。在任何应用使用 Eager Acquisition 都需要考虑这些因素。



Eager Acquisition 是 Isabella Beeton 的厨房管理格言——早做晚不做——在资源管理上的应用。

考虑到资源获取的时机，一种做法是在系统初始化的时候获取资源。这种获取策略保证对于应用来说所有的资源都不存在动态获取的问题。另一种做法是在系统初始完成之后，在应用第一次使用资源之前获取资源。比如，对于动态加载的组件，在组件加载时刻；或者在 Resource Pool（或者其他类似职能的对象）创建的时刻。不论采用哪种策略，Eager Acquisition 总是确保资源在实际使用之前已经正确获取并可用了。这样可以保证资源使用者无需任何访问开销（或者极少的固定开销）就可以使用资源了。

然而，要使用 Eager Acquisition，开发人员必须意识到这种优化实际上是一种权衡，而非只有好处。资源获取的开销并未消除，只是转移到应用生命周期的前期。如果大量的资源采用这种管理方式，那么应用、组件或者子系统的启动将会明显变慢，这对于那些要求能够快速启动的应用来说是不可接受的。Eager Acquisition 同时还有“过度获取”的风险，有些资源加载进来可能并不会用得到，这就跟这个模式的目标背道而驰了——因为这样本身就有可能导致资源耗尽。如果某些资源根本就不会用到，提前获取不会带来任何性能或者空间上的好处。

20.12 Partial Acquisition*

在实现Resource Pool (298)、Virtual Proxy (294) 的时候, 或者对象的初始化成本较高或比较复杂的时候……对象创建和资源获取这些操作可能需要满足一定的吞吐量和可预测性要求。



有些应用有严格的性能、可伸缩性和健壮性要求, 它们所访问的资源体积或者是巨大的或者是事先未知的。在系统初始化时期即获取这些资源可能会引入过度的启动开销。然而到需要时再去获取则可能会导致响应不及时。

如果内存或者处理时间方面受限, 要在系统初始化时即请求所有资源可能会导致应用的整体服务质量受到影响。甚至在启动的时候, 有些资源可能还无法访问。跟前面一样, 在真正需要时再去获取这些资源可能会导致不可预计的性能开销, 而这也可能是应用所无法容忍的。举例来说, 获取远程的资源, 不论你把它放到什么时候去做, 其开销都是很大的。

因此, 将资源获取过程分为多个阶段。在每个阶段仅获取一部分资源, 随着程序的运行逐渐取得所有需要的资源, 其目的是为了保证整体的服务质量。

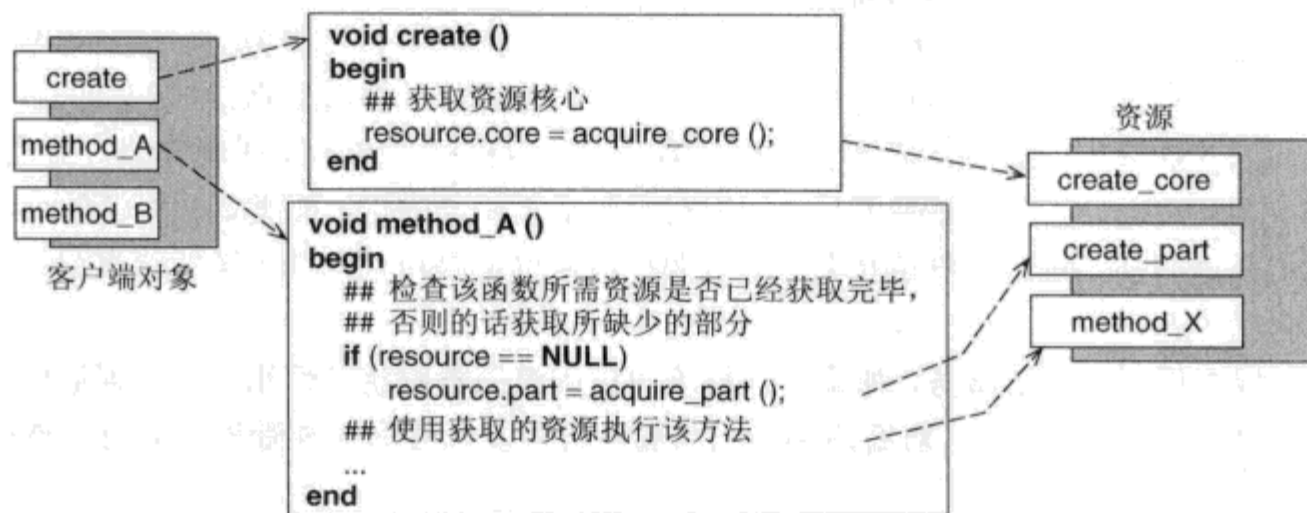


图 20-20

资源获取的成本、资源的可用性、持有或管理资源延期获取的空间开销、持有尚不需要的资源所带来的价值和问题, 以及资源获取本身所具有的自然分割特征, 这些都是我们在分割资源获取阶段时所要考虑的。例如, 远程的资源往往可以分为实例创建和实例初始化两步, 这样我们就可以方便地将资源获取也分成对应的两个阶段。



Partial Acquisition可以帮助我们既满足在资源获取的每个阶段不会引入难以承受的性能方面的代价, 又能避免在资源获取上面产生过多的开销。它并没有完全解决资源体积过大和获取时间过长的问题, 但是它从某种程度上平衡了这些问题对于其他需求的影响。

资源获取应该分为几个阶段, 在每个阶段应该获取多少资源, 以及在每个阶段应该花费多少时间, 这些都会受到多方面因素的影响。这些影响包括, 可用的内存、要求的响应时间、资源本身的状态, 以及资源的生命周期。对于有些资源来说, 我们能够做到异步获取, 即在收到请求时

仅完成初始化阶段，而在之后的某个阶段完成整个资源的获取。Future (223) 模式可以用于管理这种异步获取。

在完成部分获取之后，应用在操作资源的时候假定资源已经全部取得并初始化完成。Partial Acquisition策略常常建立在Lazy Acquisition (300) 和Eager Acquisition (301) 之上。然而，一般来说Partial Acquisition在实现上要比Lazy Acquisition和Eager Acquisition更为复杂，并同时具有这两个模式的职责。

20.13 Activator**

在实现Object Manager (291) 或者Lookup (292) 服务的时候，由于这些服务可能会需要释放客户端仍然在用的资源……所以我们需要一种简单的访问方式来临时释放资源。



在分布式系统中，有些类型的服务只能在客户端对其进行访问的时候才可以使用某些资源。客户端应该尽量不要依赖于服务的细节，比如服务所在的位置、服务是如何部署在网络的主机上面的，以及服务的生命周期是如何管理的。

无限制地使用通信信道、线程或者内存等资源会导致应用的整体服务质量受到影响。应用可以周期性地将被较少使用的资源从内存中清出，以便为其他必要的资源创造足够的空间。如果客户端再次访问这些被清出的资源，必须重新对其激活，这时候就需要重新创建资源，加载其状态，在服务器上重新启动，如果它还需要其他资源可能也需要重新获取。此类资源激活的过程对客户端来说应该是透明的，即在客户端看来这些资源任何时候都是可访问的。管理此类激活任务不应该是客户端的工作。

因此，根据需要来激活服务，如果客户端不再访问该服务则将其停用，以达到最小化资源消耗的目的。使用代理（Proxy）对客户端访问与服务行为及其生命周期管理之间进行透明的解耦合。

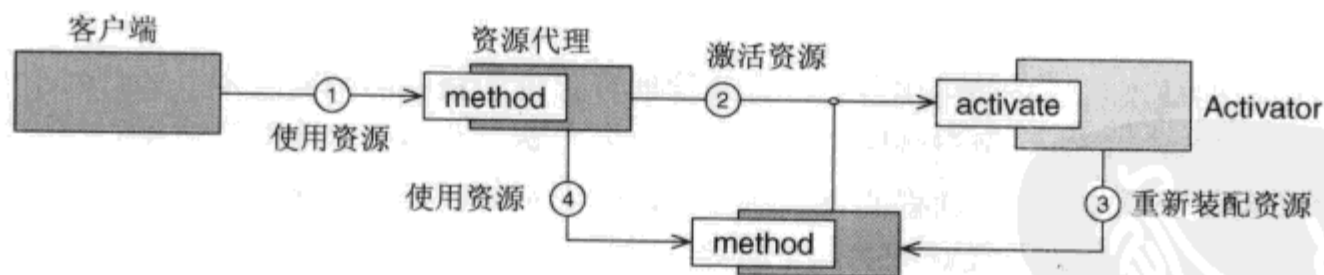


图 20-21

引入一个Activator用来完成（之前被停用的）资源的初始化和重新激活。在清出该资源之前，相关的信息发送给Activator，比如资源的ID、资源在网络中的位置、其持久化状态的位置，以及它所需要的计算资源。当客户端重新访问该（被停用的）资源的时候，Activator根据某个给定的策略，使用它所持有的有关该资源的信息将其重新激活。



Activator使得客户端不必关心怎样激活它们所使用的资源：对它们来说这些资源任何时候都是可访问的。Activator还用于保证重新激活一个资源所引入的开销不至于过大，因为它持有有关如何优化该过程的信息。比如，Activator应该在加载资源的持久化状态的同时并行地获取必要的计算资源，从而加速资源的初始化。

为了透明地使用Activator，我们需要使用Virtual Proxy (294) 等方法对资源进行包装。由于这里中介对象的引入，以及为了保存获取该资源时所需要的部分或者全部属性，该模式通常会带来一定的额外空间开销。而且，相关的资源获取方面的开销只是转移了，而没有完全消除，因此如果服务依赖的对象需要重新激活，那么其第一次执行会比其后的执行较慢。这会带来运行的不确定性。

20.14 Evictor**

在为资源受限的应用实现Object Manager (291)、Virtual Proxy (294)、Resource Pool (298) 或者Resource Cache (299) 的时候……我们需要确保能够及时地释放不经常使用的资源。



最简单的资源模型是，客户端获取一份资源，使用一次立即释放。然而，有些时候客户端可能需要多次访问同一个资源，如果每一次都经历（重新）获取—使用—释放的过程，就会引入过多的开销。但是资源提供者未必能够提供无限的资源供客户端使用。

资源的使用频率、使用时机，以及其他的使用特征都会影响资源的生命周期。约束生命周期的不仅是资源客户端的显式动作——比如资源释放，还包括整个系统环境。理想情况下，解决方案对资源客户端应该是尽量透明的，否则就会将复杂的资源管理细节推给客户端。

因此，引入一个 Evictor 监视资源的使用，并负责控制其生命周期。如果经过一段时间客户端没有访问资源，Evictor 负责将其清出，以为其他的资源留出空间。

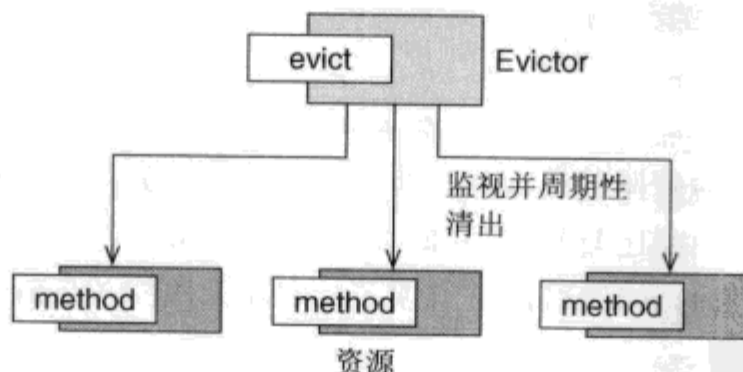


图 20-22

为每个资源添加一个标识——比如标记值 (flag)、计数器 (counter) 或者时间戳 (timestamp)，用来标记该资源什么时候使用过。在初始化的时候将其置为未使用的状态，每当使用资源的时候就对其进行更新。每隔一段时间（或者根据需要），将未标记的（或者极少使用的）资源清出，而保留标记过的资源——因为这些资源正在使用、最近使用过或者使用的频率较高。当然，我们也可以使用其他的清出策略，比如基于资源的内存消耗而不是使用频率。



Evictor将常用的（或者最近使用的）资源保留在内存中，避免了资源再次使用时重新获取的成本。同时，应用也可以控制较少使用的资源何时释放。这对于提高系统的可预测性和性能都有帮助，因为资源管理方面的活动没有和关键的应用操作掺和在一起。如果相关的资源已经被清出，应用再次访问该资源的时候就必须从头创建或者激活，当然这样就需要引入相应的性能方面的开销。

常用的清出策略包括最久未使用（Least Recently Used, LRU）和最少使用（Least Frequently Used, LFU）两种。有时候我们也会针对应用采用专门的策略。比如，在内存受限的应用中，资源的体积可以作为决定清出哪种资源的根据，对于体积较大的资源优先清出，而不论是否最近访问过。同样，有时候我们需要依据领域相关的知识做决定。比如，如果应用知道某项资源即将会用到，即使最近没有使用该资源也不应该将其清出。对于过载的反应性系统来说，Fresh Work Before Stale [Mes96]策略可以保证赋予新的任务或者请求（而不是资源）更高的优先级，而较旧的任务或者请求可能会被清出。在所有的策略中，对于有状态的资源在释放之前都应该将其状态进行持久化保存。这些策略可以是写死的，也可以使用Strategy (266) 模式做成可插拔的形式。

虽然从原则上讲非常简单，Evictor模式的微妙之处在于找到一种合适的清出策略，并且需要处理好重新激活资源和访问已经清出资源的问题。由于需要重新激活，这种方式可能使得资源的生命周期比原来略微复杂一些，引入了两种额外的关于生命周期的事件：（重新）激活和钝化（或者使其进入非激活状态）。

20.15 Leasing**

在实现Object Manager (291)、Resource Pool (298)、Resource Cache (299)、Lazy Acquisition (300)、Leasing (306)、Automated Garbage Collection (307) 或者Counting Handle (309) 结构的时候，由于在这些结构中资源分配可能跨越分布式的边界……所以我们需要确保在资源受限的应用中应该能够及时地释放资源。



如果应用运行于一个资源受限的环境中，我们需要保证相关的资源在使用之后能够及时地释放。除非资源的客户端显式地终止与其资源提供者的关系，并释放该资源，否则它们会将这些无用的资源保留在内存中。然而，如果客户端崩溃了，它就无法释放资源，甚至一些糟糕的客户端干脆没打算释放这些资源。

通常情况下，客户端要求提供者提供一份或多份资源。如果提供者允许客户端使用这些资源，客户端便可以开始使用。否则就有可能导致客户端或者提供者崩溃，或者使得提供者无法再提供某些资源，也有可能导致客户端无法正常返回某些资源。然而，如果资源提供者没有收到显式的通知说明资源不再使用，那么这些资源就会产生泄漏，这最终可能会导致资源耗尽。而且如果客户端没有收到显式的通知，表明资源不再有效，它们则可能会继续持有无效的资源。

因此，让资源提供者为客户端所持有的每一份资源创建一个租约。在这个租约中包含一个时间段，该时间段表明客户端可以在多长时间内使用这些资源。在这个时间段结束之后，将客户端中对该资源的引用和资源提供者中的这份资源释放。

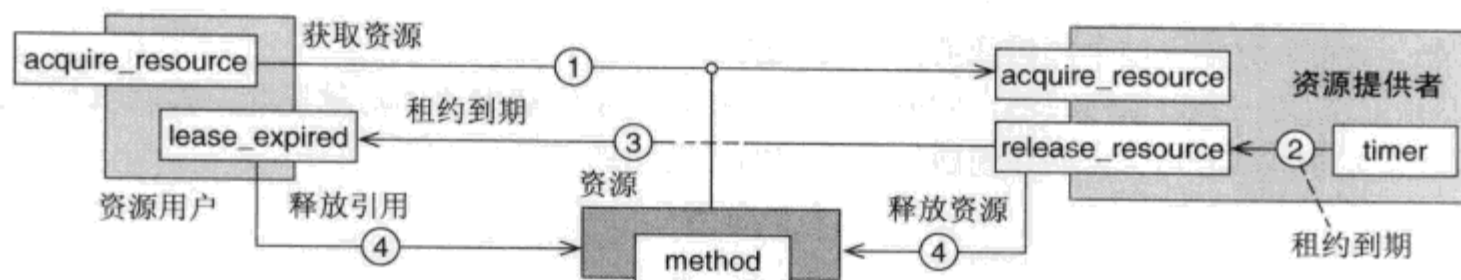


图 20-23

在租约到期之前客户端可以取消它，这时相应的资源将从资源提供者中释放。客户端也可以在租约到期之前对其进行更新（renew）。在更新之后相应的资源需要保持有效的状态。



Leasing为客户端和资源提供者简化了资源的使用和管理。客户端无需负责显式地释放资源。它们知道必要的资源在适当的时间必定是有效的，并且是可用的。资源提供者可以更有效地控制资源的使用：资源的使用被绑定到基于时间的租约之上，不再使用的资源也不会浪费，它们会被及时地释放，以便被其他的客户端使用。

Leasing背后的概念是非常简单的，其效果却非常明显，不仅提高了效率，同时还提高了系统的稳定性和可伸缩性。其他的方案——比如要求资源客户端和提供者发出周期性的检测信号或者相互侦测——往往比较复杂，而且效率也比Leasing低。总而言之，对于资源提供者和资源消费者而言，资源管理都变得相对简单，但是缺点是双方都需要依赖于时钟。如果客户端或者网络发生过载，租约可能会丢失，从而使得资源提供者无法及时收到租约更新的消息。此时，客户端使用的资源实际上已经过期，所以客户端必须能够处理这种运行时错误，比如可以获取一份新的资源。

租期的长短主要取决于应用的类型，而且往往是可配置的。为了支持分布式对象——比如在Java RMI和.NET Remoting中——默认的租期往往是几分钟。对于DHCP发放的IP地址，其默认值则是以天计算。而对于软件许可，租期则往往是以月为单位。

20.16 Automated Garbage Collection**

在实现Whole-Part (183)、Active Object (212)、Immutable Value (231) 或者Bridge (255) 等设计的时候，或者为堆上动态分配的对象提供运行时支持的时候……我们经常需要一种安全而简单的机制来回收不再需要的对象所占用的内存。



堆内存是由运行时环境所管理的有限资源，在应用中动态创建的对象所使用的内存就分配在堆上。如果不能及时将不再使用的内存收回就有可能耗尽内存，导致需要借助虚拟内存页，这样就会影响到系统的性能。同样，如果手动管理过程中出现错误，也会导致内存泄漏和内存损坏。

如果对象仅是在某个其他对象或者方法中创建并使用，其管理相对而言较为简单。比如，在C++和C#中，栈上的值对象是绑定到某个范围的，在C++中我们可以使用Execute-Around Object来自动地回收堆上对象。共享的对象和那些对外关系复杂的对象最容易引入编程方面的错误，或者为了手动内存管理而做出复杂的设计。对象关系图中有可能会包含环，即一个对象指向了另一

个对象，而那个对象反过来直接或者间接地指回这个对象。

因此，定义一个垃圾回收器，它确定应用中哪些对象没有被活动对象指向，并回收其内存。该垃圾回收器的确定和回收动作都应该是自动的、透明的。

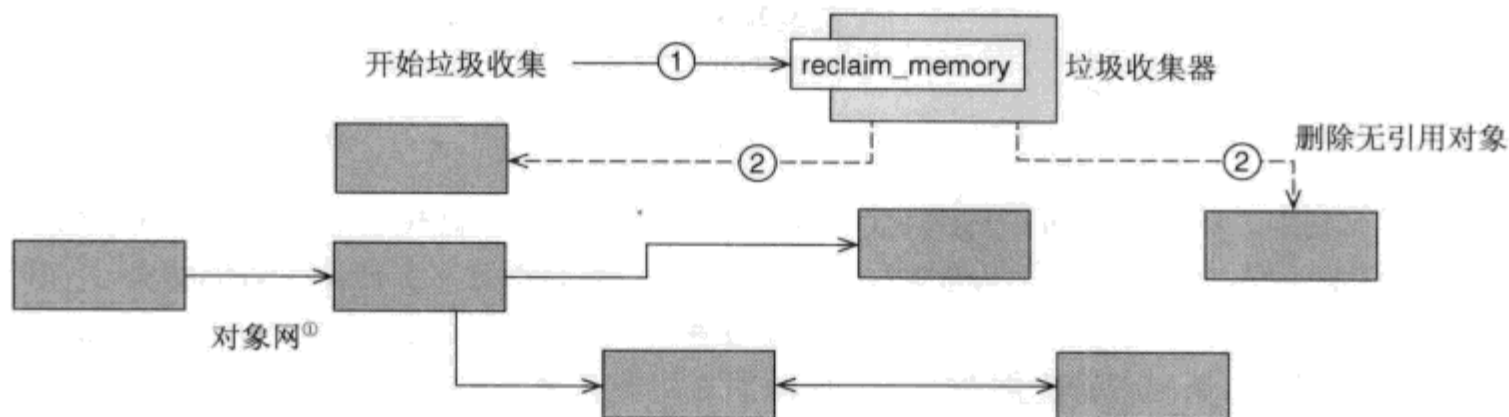


图 20-24

找到无引用对象的方法是：首先查找系统正在使用的对象，然后从所有已分配的对象中减去这一部分。所谓的根对象集合是指系统直接使用的那部分对象，比如由全局（或者静态）变量所指引的对象，以及由每个控制线程的栈所指引的对象。根据这个根集合，我们就可以按图索骥找到其他在用的对象。



Automated Garbage Collection 正如其名字所示，它将那些没有引用的对象定义为垃圾 (garbage)，无需应用的直接参与即可回收它们的内存。垃圾回收的好处是简单实用，从程序、程序员和各种类型的对象角度看均是如此，尤其是那些除了内存之外不会消耗其他资源的对象。垃圾回收要实现成确定性的 (deterministic) 是很困难的，所以服务质量可能会变化，基于资源的对象即使能够回收也不能保证及时完成。如果确实需要这样的控制，程序员需要求助于其他的手动机制，比如显式地调用 Disposal Method (314) 或者采用 Counting Handle (309)。

垃圾回收器可以同步-完整地运行，也可以以异步-增量的方式运行。最简单的方式就是“标记-清除”的办法：在第一轮遍历中标记上所有存在引用的对象，然后在第二轮中遍历内存中所有的对象，并回收未标记的对象。虽然实现起来非常简单，但是这个“标记-清除”的算法会有让“世界停止运转”的危险，这对于用户界面是非常不方便的，而对于高性能服务器或者实时系统来说是不可接受的。分代的垃圾回收器利用了对对象寿命的特征：大多数对象创建起来之后只有很短一段时间有用，而那些活过第一轮回收的对象通常会在很长一段时期内都有用。Collections for States (276) 可以将内存中的对象分为低龄对象和高龄对象，并且用不同的集合来管理它们。如果低龄对象躲过垃圾回收则被移动到高龄组。

垃圾回收器可以用各种不同的参数进行配置来对应于不同的操作行为，比如对于分代的回收器我们可以设置每一代的最大体积。通常垃圾回收API允许显式地执行垃圾回收任务，有些则支持显式地禁用和启用回收器。对于那些需要满足实时约束的应用来说能够显式地控制回收器

① 即由不同对象之间的引用构成的网状关系。——译者注

是非常重要的。不适时的垃圾回收可能会导致系统无法满足时限方面的要求，这甚至会引发灾难性故障。

在分布式环境中，如果回收策略基于对所有对象的遍历，其可行性和伸缩性都会受到影响。垃圾回收过程会因为往返传递的琐碎的消息而将网络阻塞。在同一个地址空间内进行本地的垃圾回收尽管效率比较高，但是放到分布式系统中我们还是需要Leasing (306) 这类的策略作为补充。

在垃圾回收系统中经常出现的一种资源泄漏形式是，本该回收的对象由于某些查找表中仍然包含该对象的引用而未被回收。这些表的本意只是说有那样一些对象存在，而它不会直接使用那些对象，所以并不是说那些对象就需要保留。这些表只是保留了对那些对象的引用，仅此而已。要解决这个问题，我们可以用“弱引用”作为补充手段，在查找对象的时候并不将“弱引用”计算在内。如果对象已经被回收，之后再访问这些“弱引用”对象的时候就会返回空（null）。

在使用原生模型进行手动内存管理的情形中（比如C和C++），我们也可以在运行时环境中使用垃圾回收。如果被回收对象的指针不是以特殊句柄的形式呈现，而是使用原始内存地址，我们在使用的時候就需要更加谨慎。因为原始内存地址是无类型的，所以任何一段内存只要其内容是某个已分配对象的地址，或者指向这样一个对象，我们都可以认为它是一个指针。这样的缺陷不仅会导致内存泄漏，甚至可能会对那些经过编码的地址发生误判，比如我们所看到的地址可能是偏移值而不是真实地址。

20.17 Counting Handles**

在Proxy (169)、Whole-Part (183)、Active Object (212)、Immutable Value (231)、Bridge (255)、Object Manager (291)、Automated Garbage Collection (307) 或者Disposal Method (314) 设计中……我们经常需要确保共享的对象及其资源按照一定的方式完成销毁。



在堆上创建的对象必须在使用完成之后及时销毁，否则可能造成内存或者资源的泄漏。然而在某些语言（比如C++）中，堆对象的生命周期是手动管理的。如果在生命周期管理中出现错误，就可能导致内存泄漏和内存破坏。即使在垃圾回收的环境中，资源回收也可能出现问题，因为垃圾回收并不是确定执行的。

所以，我们必须确保在堆上创建的共享对象能够可靠地、安全地、及时地回收。回收得越早，出现资源泄漏和资源饥饿的可能性就越低。然而，如果资源回收得过早，就可能会出现对对象的无效引用。只要还有一个客户端在引用该对象，它就不能回收，只有在所有的客户端都不再使用该对象之后，才可以对其进行销毁。在对象的客户端实现这种逻辑不切实际，这是因为它会使得客户端的代码与额外的处理代码混在一起，并导致客户端与对象紧密地耦合。

因此，引入或指定一个专门负责访问该共享对象的句柄对象。把跟踪对该共享对象的引用，以及在所有引用都断开之后对其进行销毁，这两项职责都封装在句柄对象里面。

句柄对象确保共享的对象能够正确地销毁，而不会产生无效引用或者内存泄漏，而且这些对于客户端来说是透明的。

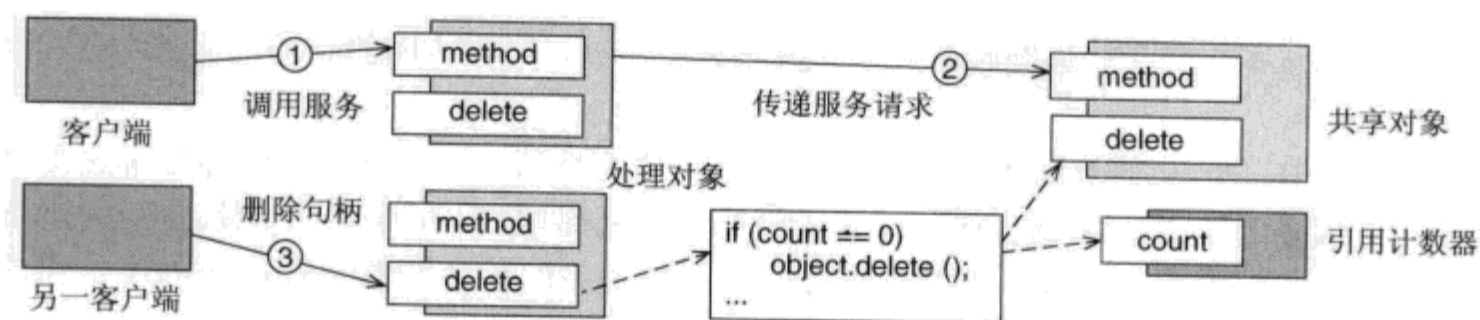


图 20-25



句柄对象的生命周期方法负责跟踪对象的引用数量，管理对象的生命周期。如果对象上没有引用了，句柄自身就可以销毁或者绑定到其他对象上，此时我们就可以将共享对象（当然这时候也算不上是“共享”了）销毁了。在C++中，相关的操作包括句柄的构造器、析构器和赋值操作符。

实现句柄对象有两个基本的方式。其一是Explicitly Counted Object [Hen01b]，它通过引用计数来跟踪共享对象上的引用，即存在一个真实的计数器。与之相反，Linked Handles [Hen01b]在句柄对象之间引入了双向的连接，这样它们就可以知道存在哪些共享对象和其他指向该共享对象的句柄对象。然而，我们很难实现一个线程安全的Linked Handles，那样在效率上通常是不可接受的，所以对于在线程间共享的对象这个方案并不适用。

通过引入一个显式的计数器，我们就不再需要将Counting Handle联系起来，而且也可以很容易、很直观地检查共享对象的引用数量是否达到上下限。我们可以很简单地判断出Counting Handle是不是共享对象上最后一个引用，并且如果是的话就在自己删除的时候将共享对象删除。

为Explicitly Counted Object实现引用计数有3种主要方式。其一是将Embedded Count直接放到共享对象内部[Hen01b]。这个方式从时间上和空间上看都是高效的，它只需要在堆上分配一次空间，相对于没有计数的共享对象并没有多占用多少空间。如果无法使用Embedded Count，或者实现起来很困难，比如我们没有源代码的访问权限，我们可以引入一个Detached Count，这个模式是通过一个单独的对象来持有共享对象的引用计数[Hen01b]。Detached Count并不影响共享对象的类型，它完全由Counting Handle进行管理。在Counting Handle开始对共享对象进行计数时创建Detached Count，此时共享对象成为Explicitly Counted Object。在最后一个Counting Handle销毁共享对象的时候，它同时也将Detached Count销毁。

第三种方案是Looked-Up Count。这种方案将共享对象的管理和计数集中在一个单独的对象里面，使用共享对象的某种ID作为键值，Counting Handle通过这个键值来访问共享对象[Hen01b]。Looked-Up Count会引入查找方面的额外时间开销，但是如果我们需要在所有共享对象上完成一些集合操作的话，这种模式也为我们提供了方便。

在多线程应用中，引用计数必须实现为线程安全的，以避免由于竞争状态而破坏对象的状态，因为有可能出现来自多个Counting Handle的并发的访问。除非计数增减的情况非常之少，否则基于锁的解决方案从时间上和资源上看都是不实用的。现代操作系统所提供的增减操作不需要锁机制，但是需要注意的是，尽管如此，这些操作还是会有一些额外的开销的，尤其是多处理器系

统或者多核处理器的情况下更是如此。

在分布式环境中，如果一个地址空间内的共享对象其生命周期由另一个远程地址空间内的对象管理，那么同步的引用计数效率就太低了，而且往往其伸缩性也不符合要求。增减的动作可能会导致网络上充满大量无用的琐碎信息，而对于客户端来说可能会导致其崩溃，或者难以遵循引用计数协议，最终可能会破坏计数机制。要在这样的环境中引入Counting Handle，我们需要使用其他的生命周期管理机制作为补充，比如Leasing (306)。而且Counting Handle (309) 对于包含环的对象关系无能为力。如果出现环，那么引用计数永远不会变成0，这往往意味着内存泄漏。

20.18 Abstract Factory**

在实现Domain Object (121)、Reflection (114) 架构或者Builder (312) 的时候……我们经常需要将相关的实现类的细节与它们的客户端接口隔离开来，以保证系统的低耦合。



客户端——尤其是那些处于某个框架内的客户端——通常都不会关心它们所使用的对象的细节，包括这些对象是怎样创建、配置、表现和销毁的。它们只关心这些对象提供了哪些服务，也就是说它们只要求这些对象在语义上是兼容的。

在很多框架（如图形用户界面工具箱、ORB等）和要求高灵活性的应用中，我们经常需要创建一族对象用以封装软件中的各种变化，比如控制流扩展(control-flow extension)、算法和数据表现等。独立地创建每个对象如果这些对象出现语义不兼容的问题，可能会给配置带来极大的麻烦。

因此，定义一个工厂接口，用来创建和销毁相关的对象系列。为这些接口实现不同的具体工厂，根据其客户端的要求执行具体的创建工作。

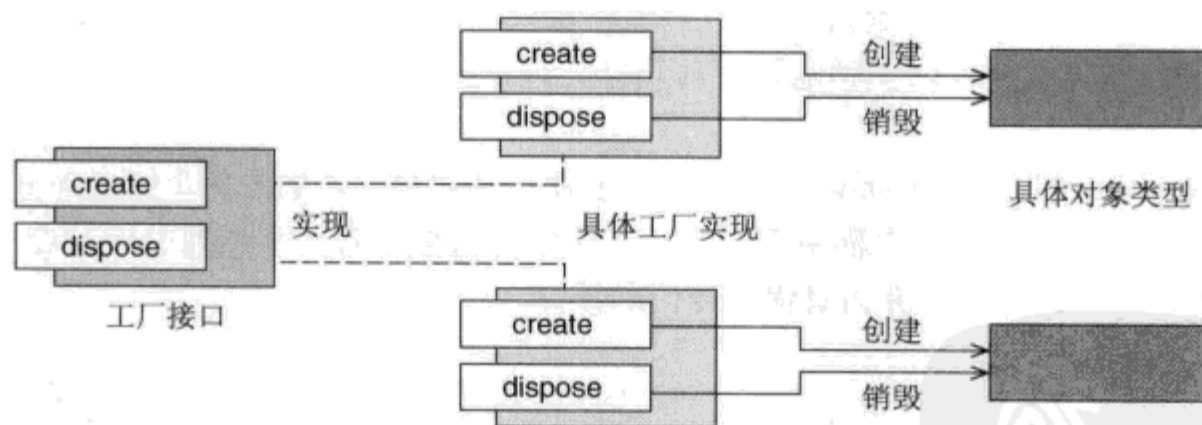


图 20-26

抽象工厂为所有相关的对象类型指定了统一的对象创建和销毁接口。具体工厂类从抽象工厂派生而来，用于控制具体对象类型的创建和销毁，确保创建的对象在语义上是兼容的。



简单地说，工厂将对象的生命周期管理同对象的使用隔离开来，使得客户端不必关心它们所使用的对象是如何创建和销毁的。同时，它们也不必依赖于对象的结构细节，这些只跟对象的创建和销毁有关。

抽象工厂接口包括两种类型的方法：一个或多个Factory Method (313) 用来创建对象，以及相应的Disposal Method (314) 用于销毁对象，后者不是必备的。我们通常都用单独的类来实现抽象工厂的职责而不把这些职责赋予已有的类，这是因为创建出来的对象大多数不具有继承的关系。

如果创建对象的具体过程存在变化，我们可以通过对工厂的配置来解决，不同的配置对象指定了工厂创建方法中真实的创建逻辑。Pluggable Factory [Vlis98b][Vlis99]就是一个例子，它实际上是一个框架，框架中其余的部分通过插入的方式提供一种简单而更为灵活的对象创建方法，这样我们就不必重复地创建过多的具体工厂了。可插入的具体行为可以通过Strategy (266) 对象或者类型实现。此外，我们还可以为工厂创建的每个对象类型提供原型实例。同态 (homomorphic) Factory Method是一种特殊形式的Factory Method，它的产品类型和负责创建的对象类型是相同的，它所表达的就是多态复制。

20.19 Builder*

在实现Domain Object (121) 或者Reflection (114) 架构的时候……我们经常需要创建一些复杂的对象，有时候这个创建过程是分几步完成的。



有些对象非常复杂，无法通过一步完成对象的创建。创建过程需要分几步完成，每一步创建对象的一部分。还有一种情况，对象的表现 (representation) 虽然简单，但是它的状态需要通过多步计算来确定。而这个步骤根据不同的对象又有所不同。

客户端通常并不关心对象创建过程的复杂性和多样性。这个过程细节不应该产生过多的临时对象或者导致应用的逻辑过于复杂。客户端只是需要一个“产品”，它要求这个产品的创建过程没有问题，而且可以正确地访问其功能。对象的创建和对象的销毁应该集中在一个地方。

因此，引入一个建造者 (builder)，它负责创建和销毁复杂对象的每个组成部分，并为其提供相应的方法，或者，将构建过程中每一步的变化组织起来。有一个单独的指导者 (director) 实现对象的创建和销毁算法，并为其提供专门的接口。

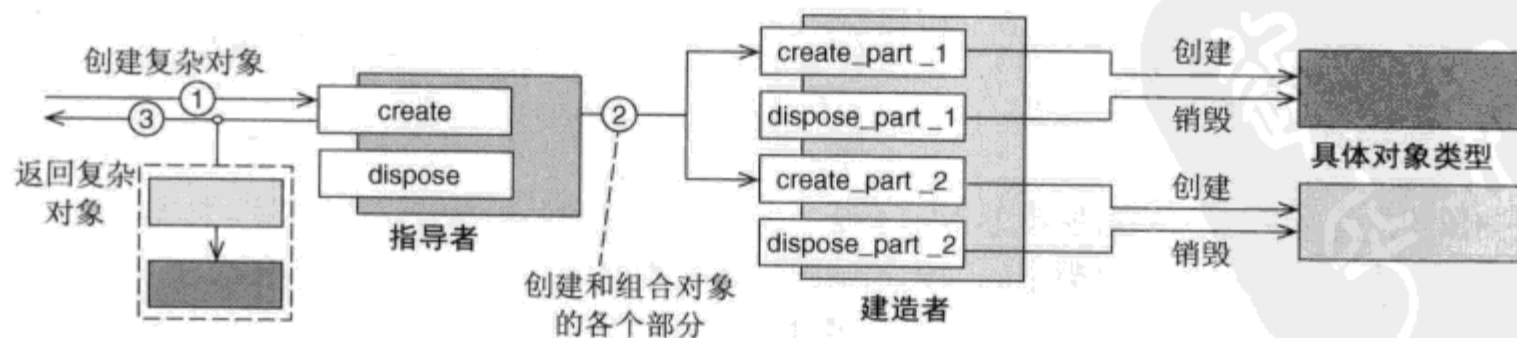


图 20-27

建造者封装了有关如何构建一个产品的知识，它或者封装了生产产品的流程，或者封装了产品的组装结构。建造者提供了一个用于返回产品对象的方法。指导者使用建造者对象来构建合适

的产品对象。该模式也可以支持对象的销毁。



将构建职能分成建造者和指导者两个角色，提高了对象创建（和销毁）流程的灵活性。建造者通过Factory Method (313) 和Disposal Method (314) 提供并封装了创建和删除对象不同部分的基础结构。指导者可以是一个专门的客户端，也可以是一个工厂（比如Abstract Factory (311)），它需要实现以某种方式组装和拆毁对象的策略，这个策略可能依赖于客户端的输入，也可能依赖于整个应用环境。建造者和指导者均可以独立地变化：只要建造者的接口保持不变，其实现的变化不会影响到指导者，反之亦然。

构成层次结构关系的对象可以通过引入类似结构关系的建造者来完成创建和删除。一个抽象的建造者声明了对象层次结构中各个元素的构建和销毁的接口，具体的建造者从这个抽象的基类派生而来，实现了特定对象的构建和销毁步骤。客户端可以通过指导者从建造者取得创建的对象，也可以将对象传递给指导者完成销毁工作。

Mutable Companion [Hen00b]是一种简单形式的Builder，它可以用更少的开销创建Immutable Value (231)，却比常规的构造器支持更多的创建方式。Java中StringBuffer和StringBuilder通过串联的方式构造String就是一个例子。值建造者在构建不变值（immutable value）的时候，避免了复杂的表达式，使得在处理过程中尽量少产生临时对象。我们可以通过修饰器方法（modifier methods）来管理累积的或者复杂的状态变化。在多线程环境中，这些方法应该具有Thread-Safe Interface (224)。客户端可以通过Factory Method (313) 来访问最后生成的不变对象（immutable object）。

20.20 Factory Method**

在实现Broker (137)、Acceptor-Connector (154)、Explicit Interface (163)、Iterator (173)、Thread-Specific Storage (228)、Immutable Value (231)、Object Manager (291)、Component Configurator (289)、Lifecycle Callback (295)、Abstract Factory (311) 或者Builder (312) 等配置的时候……我们经常需要对对象的创建细节进行封装，以保证设计的低耦合和稳定性。



有了类，创建一个对象通常并不是一件很难的事情，只要调用 new 表达式就可以了。类可以负责分配内存并初始化相应的构造器。然而，并非所有的对象构造起来都是这么简单。例如，对象的类型可能依赖于其他对象的类型，或者在构造函数之外可能还需要一些额外的初始化步骤。

直接的对象创建可能不经意间导致代码的混乱并影响调用端代码的独立性，特别是如果正确创建对象所需的某些资源尚未准备好的情况下更是如此。在发起调用的时刻，具体类和全部的构造参数可能还处于未知的状态，或者某些约束条件尚未满足。要求这些条件全部满足可能会增加调用端代码的复杂性，使其可重用性降低，在创建细节上引入变化也变得更加困难。

因此，将对象创建的具体细节封装到一个工厂方法中，使得客户端不必关心如何创建对象。

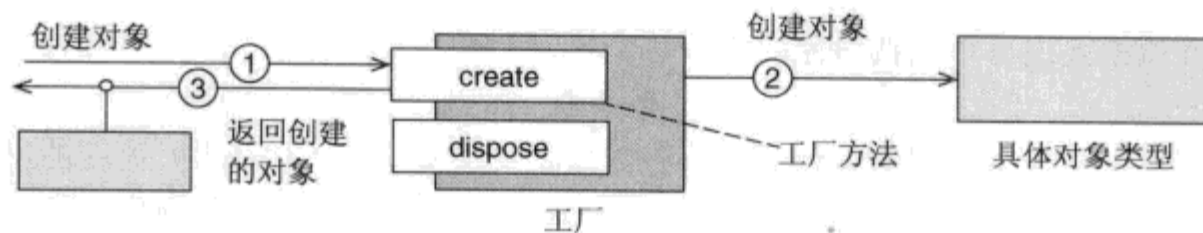


图 20-28

客户端通过调用工厂方法获得新创建的对象。



工厂方法使得客户端不必关心它们所使用的复杂对象的创建，因此客户端更容易理解和维护。对于有些对象，其初始化逻辑较为复杂——比如，需要进行额外的验证或者需要建立与某些对象之间的关系，这些并非对象本身直属的责任——无法简单地放到构造函数中实现，使用 Factory Method 模式可以简化其初始化过程。

针对不同的创建需求，Factory Method 目前有3种比较常见的变体。

- 简单工厂。将 Factory Method 放在具体类里面，它封装了对象的创建逻辑和相关的策略。
- 多态工厂。如果可能的产品类型来自某个类的继承结构，而创建哪个类型的对象是由某个已经存在的类决定的，我们可以将 Factory Method 放到这个已经存在的类里面，并在里面封装相关的具体类的知识。
- 类工厂。专门设计一个类（元对象）负责创建对象的实例。提供一个类级别的工厂方法——在很多语言中表现为 static 方法——让这些方法扮演构造器的角色。

根据不同的目标和背景，我们可以在设计中选择或者组合使用以上的这些做法。

注意，除非是有意识地在设计中引入这样一个角色，否则在接口中插入工厂的角色有时会损害对象的内聚性。这种设计的封装性固然比原来有所提高，但是相对于不使用 Factory Method 而言，其内聚性则有所降低。

20.21 Disposal Method**

在实现 Explicit Interface (163)、Iterator (173)、Object Manager (291)、Component Configurator (289)、Lifecycle Callback (295)、Automated Garbage Collection (307)、Abstract Factory (311) 或者 Builder (312) 的时候……我们需要在舍弃对象的时候，维持对对象创建和资源获取的封装。



销毁一个对象有时候并不简单，不是说显式地删除或者干脆留给垃圾回收器处理就可以了。如果我们对对象的创建进行了封装，销毁的时候仅仅回收其所用的内存可能是不够的。依赖手动的删除可能会引入过度的耦合，而垃圾回收则不能保证正确的执行结果。

对于手动内存管理而言，比如在 C++ 中，或者使用 Counting Handle 的时候，如果对象的创建是经过封装的，想当然地将其删除可能会出问题。因为其内存分配过程可能做了某种程度的优化，其分配过程和删除过程可能并不对称；而且，对于某些对象，其创建过程非常消耗资源，我们可能会采用循环使用的方式，而不是直接删除。对于此类的功能，如果由应用负责可能会导致代码

过于复杂，并且会引入与对象的生命周期管理之间不必要的耦合。

因此，将对象销毁的细节封装在一个专门的方法里面，而不是由客户端负责删除或者销毁对象。

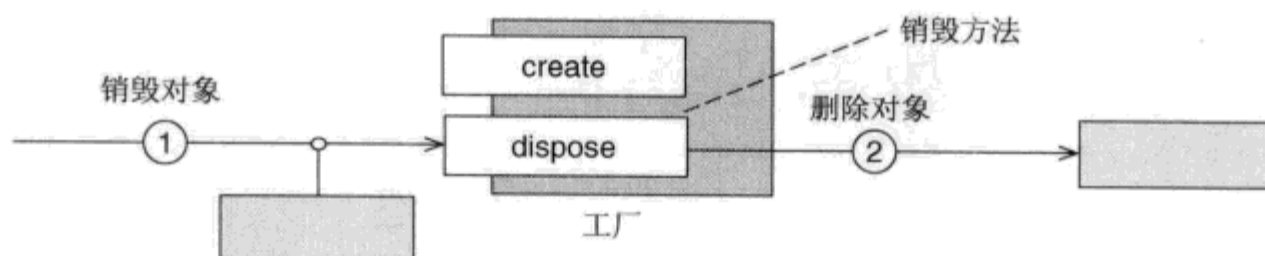


图 20-29

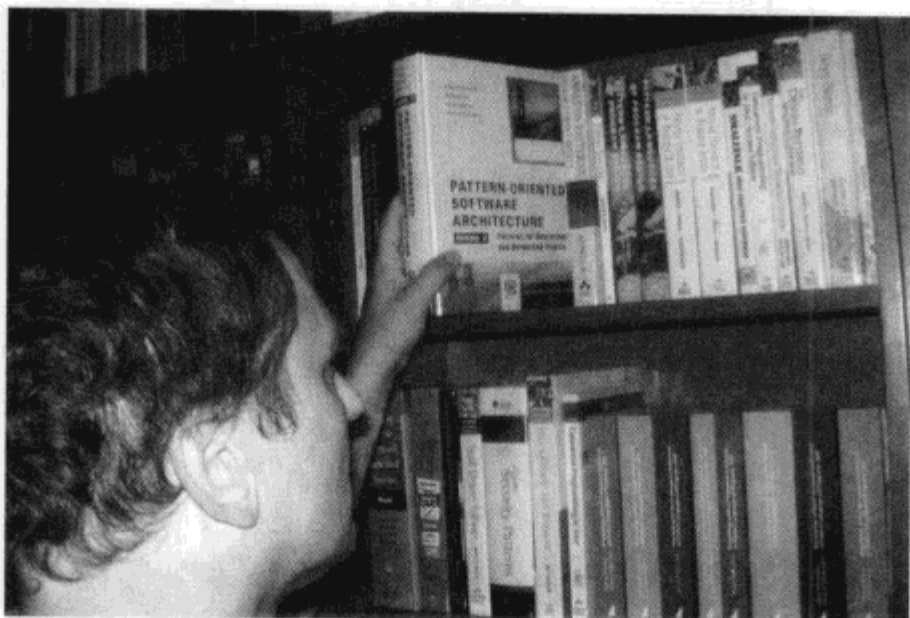
在客户端完成对象的使用之后，客户端——或者某个辅助机制——调用销毁方法，确保对象以合适的方式进行处理。可能是直接删除，也可能需要某些更为复杂的拆除工作，或者循环利用。



Disposal Method使得客户端不必关心怎样去删除或者拆除自己所使用的复杂的对象，因此客户端更容易理解和维护。而且，有些对象其本身或者某些部分需要循环利用或者供其他客户端使用，其销毁逻辑非常复杂，以至于无法在其析构器或者垃圾回收中实现，Disposal Method模式简化了其销毁过程。Disposal Method模式使得客户端不必关心对象删除行为上的变化。

Disposal Method有两种主要的实现方式。其一是工厂Disposal Method，将销毁方法放到最初负责创建该对象的工厂或者对象中。这种实现方式的好处是，被创建对象的生命周期清晰地列在工厂中。但是，这也意味着对于被创建对象需要知道它是由谁创建的，以便在使用完成之后把它交回给工厂对象。另一个实现方式是self-Disposal Method，它的销毁方法就在被销毁对象上提供。在这种实现方式中对象生命周期的关系不是那么明显，而且如果在销毁的时候对象需要返回到某个“池”中的话，对象还是需要一个对工厂的引用，以便它能够返回其“创建地”。

手动调用Disposal Method非常容易出错，所以我们往往将其包装在Execute-Around Method [Hen01a]或者Counting Handle (309) 中。在C#中，IDisposable接口提供了一个通用的自销毁接口，可以与using控制流结构搭配使用。



Frank站在书架旁
© Frank Buschmann

在很多商用系统中，它们的部分或者全部数据需要持久化保存，分布式系统也不例外。然而，在关系模型（主流的持久化范型）与面向对象模型（常用的分布式应用开发范型）之间存在着不匹配的问题。本章所展示的5个模式正是为这两个模型架起沟通的桥梁，以支持面向对象应用和关系型数据库之间的映射。

很多软件系统使用数据库来存储它们的持久化数据。在这些系统中，它们的数据库多数都是遵循了关系模型，究其原因主要是经济方面的因素。

- 已有的遗留数据和IT基础设施。长期以来数据库都是大多数IT组织必备的基础设施之一。所以，对于他们来说，要迁移到一种新的数据库模型之上，甚至即使是换一种新的关系型数据库都会得不偿失，就算新的数据库在技术上领先也无济于事。
- 客户支持。因为关系型数据库及其生态系统的成熟，它已经在全世界范围内获得广泛的支持。而这方面的服务对于其他的数据库模型来说仍显不足。
- 经验。开发人员、管理者和数据库用户最熟悉的往往是关系型数据库模型，他们在使用关系型数据库进行应用设计和开发方面，以及设计关系型数据库架构和调试此类数据库

方面积累了大量的经验。要想在其他数据库模型上获得类似经验不仅成本昂贵，而且需要大量的时间。

- 当然，还有两个重要的技术方面的原因促使我们使用关系型数据库。
- 性能。过去的时间里，我们投入了大量的精力来优化关系型数据库。所以对于大多数需要对数据进行持久化的应用来说，它们都可以提供较好的性能。
- 数据应用情景。关系型数据库支持集合和基于查询的关系模型，而在应用中数据和对数据的操作往往也符合这种模型，所以它们能够配合得很好。换句话说，多数的应用不论数据在其内部是如何表现的，它们都是使用并处理数据记录，并且在数据记录集合上完成一些操作。这个概念与关系型数据库中的概念是吻合的。

虽然一直以来关系模型都是最主流的数据库范型 (database paradigm)，然而在过去的几十年里，有关应用的设计和实现技术却发生了巨大的变化。二十年前，人们编程的方式主要是面向过程的，然而今天几乎所有的应用都是面向对象的。由于面向对象中的一些特性——比如继承、多态，以及对象间的关系——无法简单地映射到关系型数据库架构之上，这种编程方式上的变化必然带来一些不匹配的状况。

为了处理关系型模型和面向对象模型之间的不匹配，人们总结出很多模式和模式语言 [BW95][KC97][Kel99][Fow03a]。由于我们主要是关注构建分布式系统，所以这里不能详细地一一介绍。考虑到关系型数据库在众多分布式系统中的重要地位，我们为大家介绍下面几种关键的模式，以期对象模型和关系模型之间的鸿沟 (chasm) [BW95] 架起一座桥梁。

- Database Access Layer (数据库访问层) 模式 (318) [KC97] 通过在两者之间引入一个映射层将面向对象应用设计同关系型数据库分离开。
- Data Mapper (数据映射器) 模式 (320) [Fow03a] 作为应用和数据库之间的中介，在二者之间传输数据，以期解除对象模型和关系型数据库架构之间的耦合，使得双方可以独立地升级和修改。
- Row Data Gateway (行数据网关) 模式 (321) [Fow03a] 引入了一个专门的对象作为到数据库表中单条记录的通道，而同时可以使用面向对象编程机制进行访问。
- Table Data Gateway (表数据网关) 模式 (323) [Fow03a] 引入了一个专门的对象，作为到整个数据库表的通道，而且同时可以使用面向对象编程机制进行访问。
- Active Record (活动记录) 架构模式 (324) [Fow03a] 引入了一个对象，该对象包装了外部资源中的一条记录 (比如数据库表中的一行) 的数据结构，并为其提供了额外的领域逻辑。

Database Access Layer 模式最初由 Wolfgang Keller 和 Jens Coldewey [KC97] 发布。另外的 4 个模式在 Martin Fowler 的 *Patterns of Enterprise Application Architecture* [Fow03a] 中可以找到。前面列出的一些模式在 J2EE 相关的模式文献中也可以找到对应的模式：Database Access Layer 对应于 Domain Store；Table Data Gateway 对应于 Data Access Object。

所有这些模式与其他更细粒度的模式结合在一起共同完成数据库访问，这些都属于我们的模式语言的范畴。为了简洁起见，我们只给出它们的出处，而不会详细地加以介绍。没有介绍到的模式可以参考 *Patterns of Enterprise Application Architecture*，当然其中大部分在更早的时候就已经

发布了[BW95][KC97][Kel99]。

Database Access Layer可以看作是数据库访问的根模式，其他的4个模式则描述了Database Access Layer的具体实现方式。

- Data Mapper为应用中的数据模型和持久化的表结构之间提供了完全的解耦合功能。这也是Database Access Layer模式访问层最复杂的实现方式。所以仅当数据模型和表结构之间的映射相当不直观的时候才选择这种实现方式，比如数据模型中包含环，或者数据对象中的信息分布在多个表中。
- Row Data Gateway和Table Data Gateway也是为了将数据模型和数据库表结构进行解耦合。如果应用数据模型中包含的是同质对象的集合，而且这些对象可以直接映射到相应的数据库表，则使用这两个模式最为合适。如果应用的编程平台提供对记录集的支持——比如ADO.NET和JDBC 2.0，则Table Data Gateway就是最佳选择，因为记录集已经为表中数据提供了面向对象的封装。否则Row Data Gateway可能更为合适，它在单独的数据对象和表特定行中的数据之间提供了显式的映射。
- Active Record降低了应用中面向对象数据模型与数据库中表结构之间的耦合。这个模式只是简单地将数据库或者视图的某个表中特定的行做了包装，并为其添加了业务逻辑。由于在这种模式中，表结构和业务逻辑仍然耦合得比较紧密，所以要使用这种模式应该满足以下几点：数据对象的领域逻辑比较简单；数据的表现可以直接映射到数据库的表上；数据库设计——甚至是数据库本身——在应用的整个生命周期内很少发生变化。

图21-1展示了这5个访问关系型数据库的模式是如何与我们的模式语言，以及其他的数据库访问模式和惯用法关联在一起的。

21.1 Database Access Layer**

在实现Domain Model (106) 的时候，或者实现Shared Repository (117)和Blackboard (119)架构的时候……我们经常需要将面向对象的应用同关系型数据库连接起来。



面向对象软件系统经常需要使用关系型数据库作为持久化机制，这是因为对象技术可以简化应用设计，而关系型数据库支持高效的持久化；或者考虑到经济或者历史因素不得不使用关系型数据库。然而，在这两种技术之间存在着—道障碍，因为要把对象以及面向对象中的各种关系——包括对象实例之间的关系和继承关系——恰当地映射到关系型表中并非那么容易。

尽管这两个模型在各自的领域内——从经济和技术方面都显得非常合适，然而，这两个模型之间的连接并不是那么直观。我们不希望这种落差对应用和数据库的任何一方造成不良的影响。具体地说，面向对象设计的优势在实现应用时发挥得淋漓尽致，我们不愿意看到数据库访问指令和API破坏了我们的设计。同样，数据库访问的代码也应当最大限度地发挥关系型模型的优势。

因此，在应用和关系型数据库之间引入一个专门的数据库访问层(Database Access Layer)。这个数据库访问层一端为应用提供了面向对象的数据访问接口，另一端则是以数据库为中心的实现（见图21-2）。

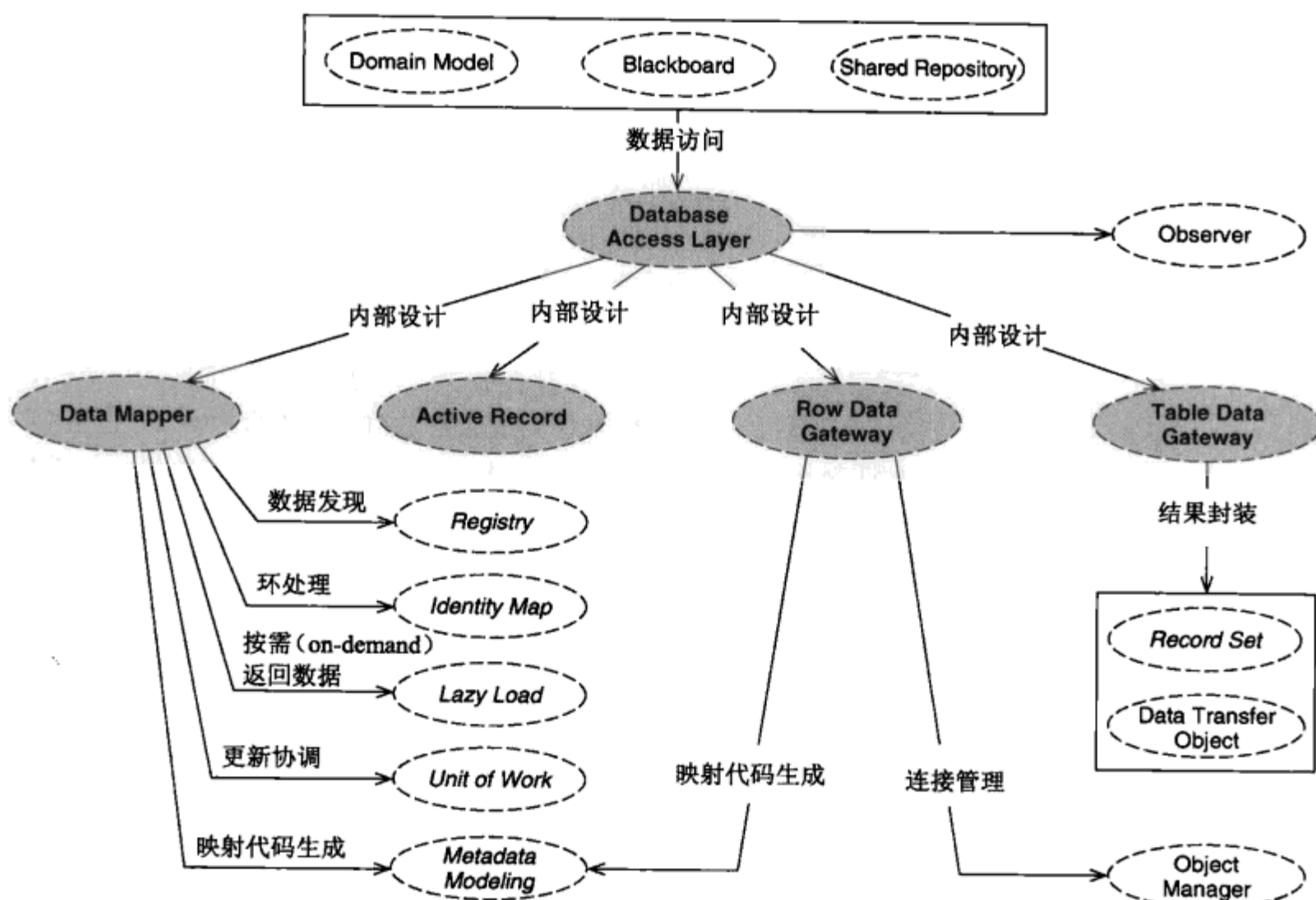


图 21-1

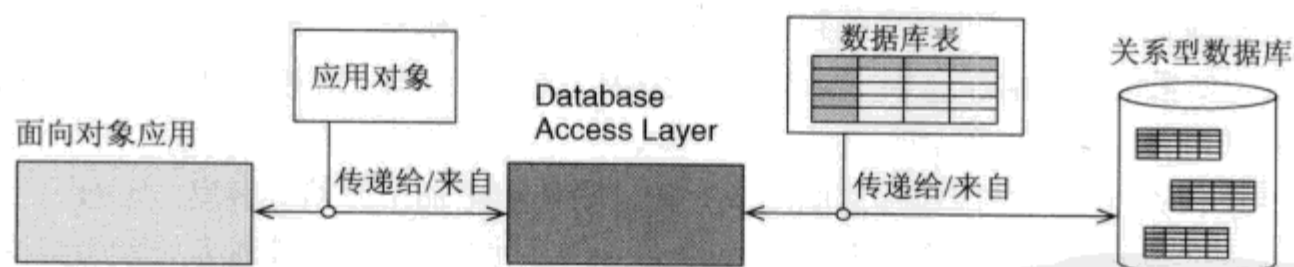


图 21-2

在不同的设计中，应用可以通过调用Database Access Layer上的方法来存取自己的持久化数据，或者让数据对象自己完成持久化。数据库访问对象用来完成在应用的数据结构和数据库表结构之间的映射。



Database Access Layer将面向对象应用和数据库的细节进行解耦合。它封装了对象和数据库表之间的映射细节，在应用看来，就像是直接存取“自己的”对象，而不是数据库表中的记录。所以说Database Access Layer为应用和底层持久化技术之间架起了一座桥梁。而且，对Database Access Layer的修改不会直接影响到应用。

Database Access Layer有很多种实现方式。Data Mapper (320) 可以让对象根本就不知道还有数据库的存在，如果应用的设计和数据库架构需要独立地升级，可以使用这个模式。Row Data Gateway (321)模式为我们提供了一个对象，它看上去就像是某个记录结构中的一条记录一样，而且它还可以通过普通的面向对象方法进行访问。如果应用的数据结构可以直接地映射到数据库的记录上，并且各条记录可以独立地访问，用这个模式比较合适。Table Data Gateway (323)跟Row Data Gateway不同，它提供了一个接口，接口中有数个find方法，用来从数据库中获取数据记录集，当然update、insert、delete这些方法也是必不可少的。这个模式适用于应用中的数据结构映射到数据库的多个记录上的情况，而且对这些记录往往是作为一个整体统一访问的。Active Record (324)是这里面最直接的访问方式，所有的数据访问逻辑都放到应用对象中。如果应用中的对象模型可以直接映射到数据库架构上，而且领域逻辑又比较简单——比如，其操作不外乎创建、读取、更新和删除——就可以使用该模式。

如果数据库被其他的应用访问所更新，则Database Access Layer也负责通知所在的应用。通常这种更新机制都是由Observer (237)模式来实现，数据库通知Database Access Layer，再由后者通知应用。缺乏这种通知机制，那么应用中数据库的视图可能会慢慢失去时效，甚至最终会影响到应用的行为。

21.2 Data Mapper**

在设计Database Access Layer (318)的时候……我们不希望应用依赖于数据在持久化存储中的表现形式。



面向对象应用和关系型数据库的数据结构采用了不同的机制。然而，我们需要在二者之间进行数据传递，如果面向对象领域模型依赖于关系型数据库的架构，则对其中任何一个所做的修改都可能影响到对方，反之亦然。

面向对象和关系型数据库架构之间的映射无疑给应用增加了新的复杂度。比如，在关系型数据库中并没有集合、继承这些概念，而在面向对象语言中也不存在SQL查询等关系型结构。当然，直接把数据库访问代码写到应用中并不难，只是增加了对象的复杂性，一旦应用或者数据库的架构发生变化，这种方式的脆弱就凸显出来了。而且直接把应用中的类翻译成数据库中的表往往不是最合适的数据库架构。所以，我们需要一种松耦合且稳定的解决方案。

因此，为每一种类型的持久化应用对象引入一个Data Mapper，其作用是在对象和关系型数据库之间传输数据（见图21-3）。

Data Mapper是面向对象领域模型和关系型数据库之间的中介。客户端可以使用Data Mapper来从数据库中取得应用对象，也可以通过Data Mapper将应用对象保存到数据库中。Data Mapper负责所有必要的数据库转换，并确保对象与数据库保持一致。

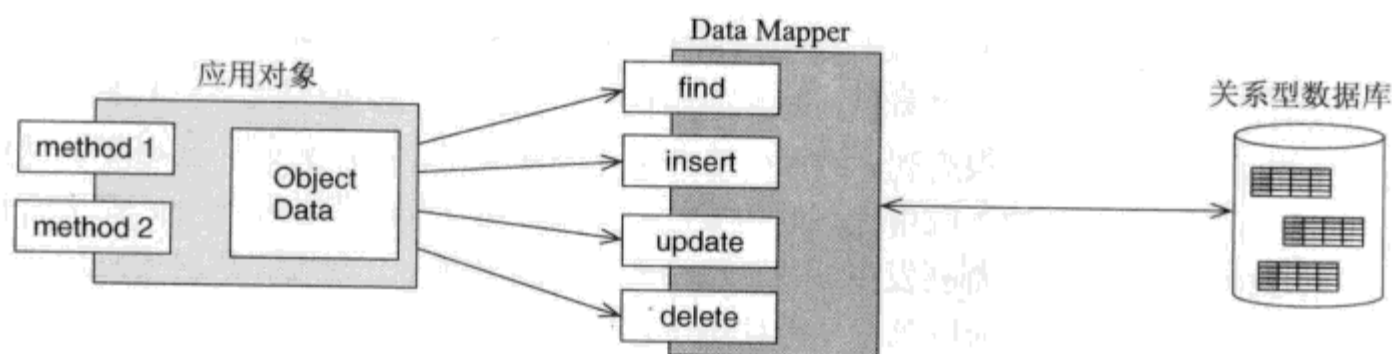


图 21-3



使用Data Mapper模式，内存中的对象不必知道还存在一个与之相应的数据库。而且，在对象中不再需要SQL接口代码，也不需要了解数据库的架构。Data Mapper模式使得关系型数据库架构和面向对象领域模型可以独立地升级。这个设计也有助于简化单元测试，原本使用真实的数据库的Data Mapper可以通过mock对象来进行模仿，以支持测试装置（test fixture）。

如果应用的领域模型非常简单，而且基本上可以与数据库中的物理表现对应起来，可以使用Data Mapper直接将数据库中的表用一个字段对应一个成员的方式实现映射。如果领域模型比较复杂，可以使用其他的模式来共同完成数据映射功能。Registry[Fow03a]模式可以用来发现属于某个应用对象的数据。如果应用对象之间的依赖关系比较复杂，甚至存在环状依赖，可以使用Identity Map[Fow03a]来确保数据只会加载一次。如果应用对象中包含大量数据，而且其中一些不太重要或者不常用的数据可以延迟加载，我们可以使用Lazy Load[Fow03a]保证Data Mapper在对象创建时只加载关键的部分数据，而其他数据在第一次访问时加载。一般来说，Lazy Load 是由Partial Acquisition (303) 和 Lazy Acquisition (300)组合而成的，它们共同完成对数据加载流程的控制和优化。在向数据库插入数据或者更新数据的时候，Data Mapper可以使用 Unit of Work [Fow03a]来获知哪些对象已经修改、哪些已经创建或者销毁。

如果Data Mapper需要支持处理来自不同类型应用对象的数据，为了避免将不同的映射机制硬编码到代码中，我们可以使用Metadata Mapping [Fow03a]模式。

Data Mapper简化了应用对象的编程和依赖关系。它提供了一定程度上的隔离和稳定，使得应用对象和数据库架构不会因对方的改变而受到影响。当然，Data Mapper自身也有一定的复杂性，不论是应用对象模型还是数据库架构的变化都会连累到Data Mapper本身的设计。

21.3 Row Data Gateway**

在设计Database Access Layer (318)的时候……我们需要一种访问和管理单条数据记录的方法。



在有些应用中，进程内的数据结构可以直接映射到关系型数据库架构上，在行和对象中间存在一一对应的关系。然而，让每一个对象直接访问其相应的行，不仅会将应用代码和基础设施代码（infrastructure code）紧密地耦合在一起，也会削弱应用代码的设计。

当看到应用的数据对象类型可以对应到数据库中的表，数据结构的实例对应到行，数据结构的数据字段对应到列时，我们很容易想到要在应用代码中直接访问和使用关系型数据，因为这样不仅内存使用得较少，代码也很直观。然而，这种设计的缺点是，应用代码和数据库访问代码紧密地耦合在一起，严重地影响了应用代码的内聚性。如果数据库架构发生了变化，在所有使用到该数据的地方相应的映射代码都要发生变化。这种变化会越来越多，应用和映射代码的复杂性也会随之增加。与此类似，如果应用支持多种使用不同SQL风格的数据库，应用代码中必须显式地处理这些变化，于是使用较少内存和代码直观等优势便荡然无存了。

因此，将数据结构和相应的数据库访问代码包装在Row Data Gateway中，其内部结构与数据库中的记录相似，但是它为客户端提供了独立于表现的数据访问接口（见图21-4）。

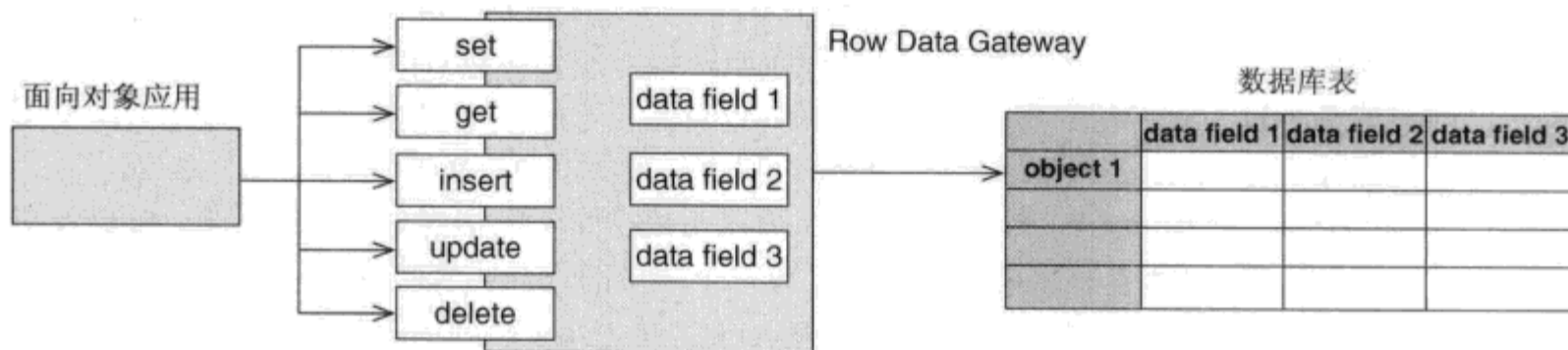


图 21-4

对应于表中的每一行都有一个Row Data Gateway实例。客户端使用这些网关（Gateway），就像使用普通的应用对象一样。在客户端创建新的网关实例时，其数据作为相应的数据库表中的一行记录插入到数据库中。当客户端向Row Data Gateway提交变更的时候，数据库表中的相应字段自动更新。在删除网关对象的时候，数据库表中相应的行也会被删除。



如果对于数据库记录的访问、管理和存储是显式的，而且是独立于数据库其他记录的，则使用Row Data Gateway最合适。每一个Row Data Gateway实例代表一条单独的记录，通常就是数据库表中的一行，但是所有对数据库访问的细节都隐藏在其接口之后了。如果Row Data Gateway需要移植到其他的数据库上，并使用不同的SQL风格，这种对表的表现的修改对客户端是透明的，同时数据库访问代码的修改对客户端也是透明的。

Row Data Gateway负责将数据源中的数据类型转换为内存中的数据类型。类型转换通常都很直观，这是因为数据库表中的每一列都对应于Row Data Gateway中的一个成员，这也意味着我们可以方便地使用 Metadata Mapping [Fow03a]来生成映射代码。

Row Data Gateway的接口非常简单：set 和 get 方法用于访问和修改内存中的数据结构，update、insert和delete 方法用执行适当的SQL来操作数据库。在Row Data Gateway中并不包含应用逻辑，其目的是保证被封装的数据结构与其上的操作相互独立。

每个Row Data Gateway对象必然是跟某种网络连接相关的。为了保证该管理代码的简单性和自我完备性，我们可以使用Object Manager来创建新的记录。这同时也解决了可能为一行记录创建多个Row Data Gateway实例的潜在问题。同时，我们也可以考虑使用Object Manager来支持行

删除，而不是通过网关对象。Object Manager还可以提供查找的功能，在数据库中找到某个表。

21.4 Table Data Gateway **

在设计Database Access Layer (318)的时候……我们必须提供某种管理整个数据集合的机制。



在有些应用中，进程内的数据结构可以直接映射到关系型数据库架构上。然而，直接在应用的业务逻辑中通过SQL管理数据集合会将应用模型和数据库架构紧密地耦合在一起。

如果应用的数据结构类型和数据库表相对应，我们很容易做到在应用代码中直接访问关系型数据，因为这样内存消耗较少而且代码也比较直观。然而，如果数据库架构需要升级或数据库技术发生了变化，由此而产生的耦合会给开发带来问题。而且，应用中会充满非内聚的基础设施代码。虽然Row Data Gateway为该问题提供了一种简单的解决方案，它也引入了其他的问题，比如行管理，同时还可能会导致对象激增。对于表比较小而且在一个会话中只会访问少数几个表的情况下，这些问题可能并不严重，但是对于大型的应用来说，这可能就很严重了。而且Row Data Gateway也没有处理对象集合，因为它只负责管理单个对象。

因此，将访问某个数据库表的代码包装在一个Row Data Gateway中，并为其提供一个接口，允许应用访问这些领域专属的数据集合（见图21-5）。

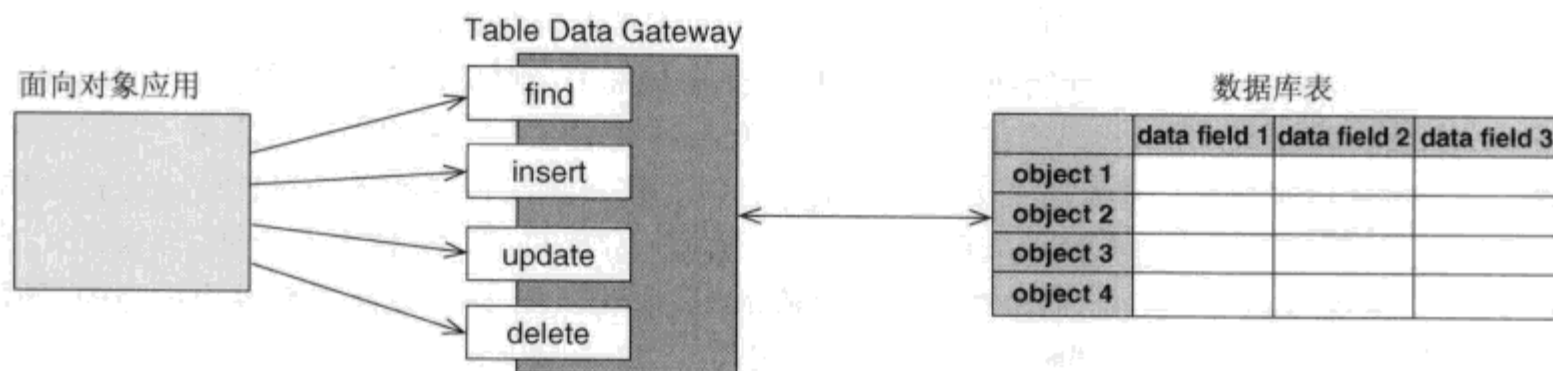


图 21-5

在数据库中的每个表都有一个Table Data Gateway实例与其对应。客户端使用该网关管理相同类型的数据集合。当客户端向集合中添加新的对象时，其数据作为一条新的记录添加到数据库表中。对数据库表中记录的修改和删除也通过网关来完成。



如果数据库的记录是以集合的方式修改和存储的，则使用Table Data Gateway最为合适。Table Data Gateway封装了访问数据库的细节，并负责数据和领域专属对象集合之间的相互转换。在将Table Data Gateway转换到使用不同SQL风格的其他数据库上的时候，数据库表的表现的变化对客户端来说基本上是透明的，同时对数据库访问代码的修改对客户端也是透明的。但是这种设计不支持对数据库架构进行大规模的修改。

Table Data Gateway的接口很简单，其中包括几个find方法，用来从数据库中获取数据，此外还包括update、insert和delete方法。这些方法将其参数和操作翻译成SQL语句，并在数据库上执行。

Table Data Gateway通常是无状态的，因为它的任务只是推拉数据。该接口的一个约束是，如果Table Data Gateway对应的是一个视图，更新和删除操作就不能使用了。

将查询结果返回给客户端有很多不同的做法，这些结果中可能会包含数据库中的多条记录。其中一个方法是返回一个简单的数据结构，比如一个map，或者Data Transfer Object (244)。在有些环境中，比如ADO.NET和JDBC 2.0，我们可以直接返回一个Record Set[Fow03a]，它是一种表格数据的内存表现。因为Record Set只是表结构在应用代码中的一个镜像，所以它仍然会使得应用代码与数据库紧密地耦合在一起。Table Data Gateway也可以以工厂和管理者的角色出现，用来返回适当的领域专属的对象。

21.5 Active Record

在设计Database Access Layer (318)的时候……对于一些自我完备的数据记录，它们只提供一些简单的处理方法，我们不想使用太复杂的数据映射。



在多数的面向对象应用中都存在一部分逻辑是与其操作的数据绑定在一起的。如果数据是保存在数据库中，那么它就不能直接操作这些数据。如果数据结构所对应的行为比较简单，使用分层的方式就可能有些小题大做了。

在应用代码中直接使用数据库的API，可能会导致代码内聚性降低，而且在需要修改的时候非常的脆弱，而像数据库架构升级或者技术迁移等往往都是不可避免的。这时我们应当避免随意地将数据库访问代码和应用代码混在一起。然而，有时候我们对数据库的操作只是对一条记录的属性（attribute）的查询，或者对几个属性的简单计算，在这种与记录相关的对象行为比较简单的情况下，如果在应用对象和数据库访问之间过度地引入分离机制，会使得设计过于笨重。

因此，使用Active Record对象对数据、相应的数据库访问代码，以及以数据为中心的领域行为进行封装，并为客户端提供一个面向特定领域的接口（见图21-6）。

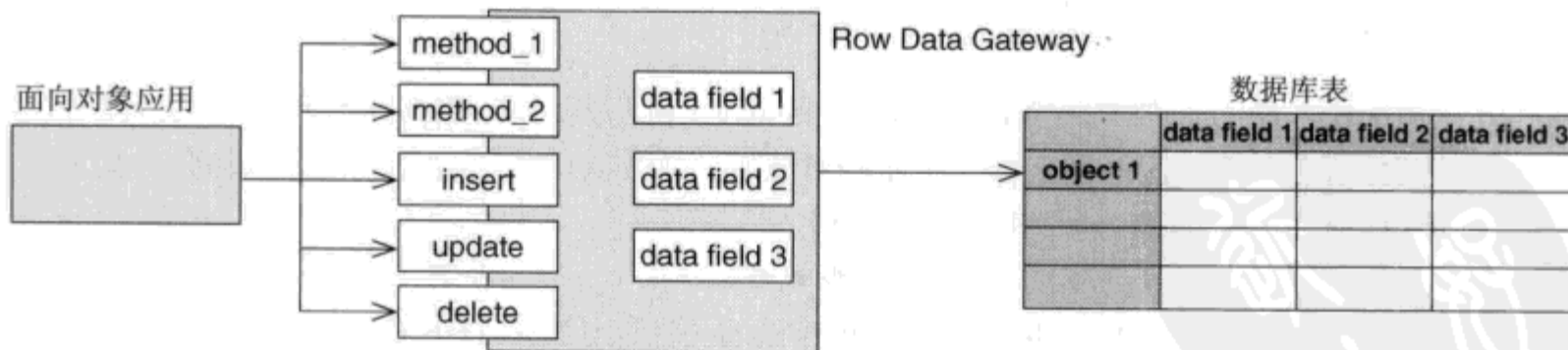


图 21-6

客户端通过其面向领域的接口访问Active Record对象，就像访问普通的应用对象一样。如果被封装对象的值发生变化，我们可以通过update方法来更新数据库中相应的行。如果客户端创建了一个新的Active Record，在相应的表中就会新添加一行。如果客户端销毁了一个Active Record，相应的表中的行也会被删除。



Active Record模式比较适合于应用的领域逻辑较为简单的情况，比如主要包括create、read、update和delete等操作，以及属性管理和基于属性的简单计算。在这种情况下，修改表结构对应用的代码所产生的影响就局限在Active Record内部，同时如果要将Active Object移植到使用其他SQL风格的数据库上对数据库访问代码的修改也限制在一个较小的范围内。如果领域对象类型较为复杂，比如使用了继承或者对象间关系或者集合对象，那么Active Record就不太合适了。

Active Record类通常都提供了从数据库领域对象创建实例的方法。这可以包括SQL结果集、static查找方法（封装了常用的SQL查询）、更新数据库和将封装数据插入数据库的方法、get/set等存取Active Record数据字段的方法，以及与领域相关的方法（这些方法实现了应用的业务逻辑中以数据为中心的部分）。然而，在Active Record设计中，如果问题领域概念和基础设施概念混在一起就会导致应用和数据库的紧耦合。



赛后即是赛前。

——德国足球名言

上面这条关于德国足球的谚语取自《赛后胜似赛前》一书。在本书结束之前，我们再来回顾一下我们的分布式计算模式语言，并且列出面向模式的软件架构（POSA）下一卷（也是最后一卷）中所要探讨的话题。

我们在本书中所展现的模式语言来自于我们自己以及世界各地专家在构建分布式系统方面的经验。我们看到了文献记载的模式背后的语言。这些模式广泛应用于各种分布式系统的开发，从工业自动化到电信和医疗成像，从基于Web的电子商务系统到组件和通信中间件。我们的语言有两个主要的意图。

- 为分布式软件系统构建中的核心领域做出综述性的介绍，并为其提供指导和交流手段。
- 将各种文献中与分布式计算有关或者对其有价值的众多独立的模式、模式的集合和模式语言联合起来，形成一个语言，为构建分布式系统提供完整而一致的视图。

虽然我们占用了这么多章节，覆盖了如此广泛的主题，但是我们的语言仍然还在完善之中，也许永远也不会有最终完成的那一天。随着我们在构建分布式系统方面的经验不断地增长，在应用这个语言的过程中不断地得到反馈，更多的来自各种文献的模式会被集成进来，新的模式会不断地涌现，已有的模式在内容方面和与语言中其他模式的关系方面也会不断地得到重构。这么艰巨的任务单靠我们不可能完成，要是那样不但这个语言完不成，恐怕我们连饭碗也保不住了！因此，我们希望你——本书的读者——和软件模式社区的其他人一道帮助我们改进这个语言，使其不断发展。我们的目标是为架构师和分布式系统的开发人员提高其实用价值。

另一个值得我们进一步研究的方向是我们这个模式语言和其他模式语言所赖以构建的概念框架。在第1章模式和模式语言中，我们简要地介绍了相关的概念，包括模式、模式序列和模式语言，然而实际上还有很多值得研究的内容我们未在本书中涉及。比如说上述概念的内在含义是什么，还有哪些相关的性质值得探索，最重要的是所有的这些概念是如何彼此联系在一起的，它们怎样为模式塑造一个一致而清晰的视图。

这些问题的答案我们放到了POSA的下一卷*Patterns and Pattern Languages*（模式和模式语言）中回答，它也将是我们的面向模式的软件架构系列的最后一卷。敬请期待！

术 语 表

本术语表给出了本书中常见的关键术语的定义。这些术语涵盖了分布式计算机软件系统的各个方面。我们用楷体表示引用到的表中的其他术语

有些术语在书中只出现过一次，在这里没有给出定义，比如企业资源计划（enterprise resource planning, ERP）。如果此类术语需要解释，我们会在文中给出，以便您可以结合上下文理解

为了保持完整性，这里也包含了一些在[POSA1]各部分中深入介绍过的通用术语，比如模式、软件架构等习惯用语

抽象类（Abstract Class）：如果一个类的接口中的部分（或者全部）方法没有实现，我们称之为抽象类。抽象类定义了其子类的公共抽象

抽象组件（Abstract Component）：如果一个组件为其他组件指定一个或多个接口，我们称之为抽象组件。抽象组件可以是显式的，比如抽象类；也可以是隐式的，比如C++模板方法里面的类参数。抽象组件构成了使用多态实现灵活系统的基础。抽象组件和抽象类常常是表示同一个概念，使用抽象组件的目的是避免将模式限制在面向对象语言上面

抽象方法（Abstract Method）：在抽象类中声明，并由子类实现的方法称为抽象方法

主动连接建立（Active Connection Establishment）：由向远程应用发起连接的一方扮演的连接角色（对比被动连接建立）

主动对象（Active Object）：客户端调用对象上的方法时，如果该对象的方法在不同于该客户端的线程上执行，我们称这个对象为主动对象（对比被动对象）

API（API）：应用编程接口。可重用软件平台——比如操作系统——的外部接口，被构建于其上的系统或者应用使用

应用、应用程序（Application）：程序或者程序的集合，用于满足客户或者用户的需求

应用框架（Application Framework）：应用框架是一套集成的组件，它们通过协作的方式为一族相关的应用提供可重用的软件架构。在面向对象环境中，应用框架包含抽象类和具体类，以及控制反转。对这样的框架进行实例化往往需要为对已有的类进行组合或者子类化

架构模式（Architecture Pattern）：架构模式展示了软件系统基本的结构组织形式。它提供了一套预定义的子系统，指定了子系统的职责，还提供了关于如何组织它们之间关系的规则和指导

关联型数组 (Associative Array): 关联型数组是指数组的索引键可以使用任何类型, 比如字符串, 而不一定是整数。散列表和二分查找树是关联型数组的常见实现

异步I/O (Asynchronous I/O): 异步I/O是一种数据发送和接收的机制, I/O操作发起之后, 调用者不会阻塞并等待操作完成

巴克斯诺尔范式 (BNF) (Backus Naur Form): 巴克斯诺尔范式是描述语言语法的一项标准技术

带宽 (Bandwidth): 通信设施 (比如网络、总线) 的容量

广播 (Broadcast): 一种特定的组播形式, 将消息发送给某个域内的所有服务器

总线 (Bus): 一种连接计算设备 (比如CPU、硬盘、网络接口) 的高速通信信道

忙等待 (Busy Wait): 线程等待锁的一种方式。线程通过执行紧凑循环的方式轮询在每次迭代中检查锁是否有效。与之相对的另一方式是, 线程休眠并允许其他线程继续执行, 直到锁被释放

缓存绑定 (Cache Affinity): 一种线程调度优化。CPU刚刚执行过的线程其状态可能仍然保存在CPU的指令和数据缓存中, 优先将最近执行的线程提交给CPU可以提高这种可能性

回调 (Callback): 回调指的是一个方法或者对象, 它用于指定在某个特定事件发生时应该执行的动作

类 (Class): 类是面向对象语言中的基本构件。类指定一个接口, 封装其实例或者对象的内部数据结构 and 功能。类通过继承可以扩展一个或多个父类, 由此产生的类称为子类

客户端 (Client): 在我们的描述中, 客户端指的是一种角色、组件或者子系统, 它调用或者使用由其他组件提供的功能

关闭 (Closure): 参见方法关闭

协作者 (Collaborator): 与其他组件协作的组件。同时也是CRC卡片的一个元素

归置优化 (Collocation Optimization): 中间件中使用的一种技术。当发送者和接收者搭配时, 即通信双方处在同一进程或者同一台计算机上时, 该技术可以消除 (解) 封送的数据或者传输请求响应的不必要的额外开销 (对比分布)

完成事件 (Completion Event): 一个包含响应信息的事件, 该响应信息与由某个客户端发起的请求事件相关联

组件 (Component): 软件系统中一个完备的、可部署的、可执行的部分。组件为其他组件或者客户端提供某种甚至一整套服务。组件包含一个或多个接口供别人访问其服务。组件是构造系统的构件块。虽然组件是自完备的, 它仍然可能由其他组件组成, 或者依赖其他组件。在编程语言这个层次, 组件可以表现为模块、类, 甚至是一组相关的函数

组件对象 (Component Object): 一个对象, 其类的定义封装在一个已部署的组件内。组件对象所实现的接口由方法组成, 而不暴露实现。一个组件对象的类封装在组件里面

具体类 (Concrete Class): 其对象可以实例化的类。与抽象类相反, 具体类中所有的对象都有实现。该术语用于区分具体子类与其抽象父类

具体组件 (Concrete Component): 具体组件其接口定义的所有元素均有实现。用于和定义其接

口的抽象组件区分开来，它们的关系与具体类和抽象类的关系一样

并发 (Concurrency): 并发指的是对象、组件或者系统执行“逻辑同步”的操作的能力（对比并行性）

条件变量 (Condition Variable): 条件变量是一种同步机制，协作的线程使用这种机制临时挂起，直到包含线程间共享数据的预期条件得到满足[IEEE96]。条件变量总是和互斥量一起使用，线程在判断条件之前必须先取得互斥量。如果条件判定为假，则线程自动挂起在该条件变量上，并释放互斥量，以便其他的线程可以修改共享数据。如果协作线程修改了这个数据，它可以通知条件变量，从而自动地继续先前挂起在该条件变量上的线程，并再次取得互斥量

连接 (Connection): 一种用于在网络应用终端之间交换数据的全关联

容器 (Container): 容器是持有某种元素集合的数据结构的通用名称。list、set和关联型数组 (associative array) 都是容器的常见例子。而Enterprise JavaBeans、ActiveX控件和CORBA组件模型 (CORBA Component Model) 也定义为容器，因为它们为其他组件提供运行时环境，并且使得其他组件不必关心操作系统之类的底层基础设施

CORBA (CORBA): 通用的对象请求中间人架构 (Common Object Request Broker Architecture)，一种由Object Management Group (OMG) 定义的分布式对象计算中间件标准

CPU (CPU): 中央处理单元。一种执行二进制程序指令的硬件组件

CRC卡 (CRC Card): Class-Responsibility-Collaborator (类职责协作者) 卡。一种描述软件架构中类的职责和协作者的设计工具和概念

临界区 (Critical Section): 对象或者子系统不应并行执行的代码可以通过临界区进行同步。临界区是一个指令序列，它满足以下特征：当一个线程或者进程正在临界区内执行时，其他线程或进程不可以在里面执行（对比读端临界区和写端临界区）

数据缓存 (Data Caches): CPU中设置的特殊的高速存储，可以提高系统的整体性能。缓存持有主内存的一部分副本，所以对应用来说就跟正在访问主内存是一样的

数据模式套接字 (Data-Mode Socket): 参见套接字

死锁 (Deadlock): 死锁是一种并发的风险。如果多个线程视图获取多个锁，因而进入无休止的循环等待状态，就称为死锁

解封送 (Demarshaling): 将一个封送消息从系统和应用独立的格式转换成系统和程序特定的格式

分离 (Demultiplexing): 分离是一种将从输入端口传入的事件路由到目的接收者的机制。在输入端口和接收者之间是1对N的关系。分离经常用于传入事件和数据流。其相反的操作称为“多路复用”

设计 (Design): 软件开发者所从事的一种活动，其结果是系统的软件架构。该术语也用于表示这个活动的结果

设计模式 (Design Pattern): 设计模式提供了一个精炼软件系统中的元素，以及这些元素之间关系的方案。它描述了经常重现的角色交互的结构，解决某个特定背景下的某个设计问题

开发属性 (Developmental Property): 与开发相关的非功能属性，包括可维护性、可扩展性、

适应性和重用性。通常，开发质量对于系统用户是不可见的，客户一般对其不感兴趣。主要是软件开发机构会关心软件系统开发质量，并受其影响。（对比操作性质量。）

设备 (Device): 设备指的是在计算和通信系统中提供服务的硬件组件

设备驱动 (Device Driver): 设备驱动是操作系统内核中的一种软件组件，它负责控制连接到计算机的硬件设备，或者像虚拟磁盘这样的软件设备

分布 (Distribution): 分布指的是将对象和访问它的客户端置于不同的进程或者主机的行为。分布往往用于提高容错能力或者访问远程的资源（对比归置）

分布式系统 (Distributed System): 分布式系统是一种计算系统，系统中多个组件通过网络通信的方式相互协作。自从二十世纪九十年代中期，因特网和万维网的爆炸性增长使得分布式系统从传统的工业自动化、国防和电信领域扩展到了几乎所有的领域，包括电子商务、金融服务、医疗保健、政府部门，以及娱乐业

领域 (Domain): 领域是表示与特定问题相关的概念、知识等。经常用于“应用领域”来表示与某个应用相关联的问题。在因特网上，域表示一个逻辑位置实体，比如uci.edu或者siemens.de

领域分析 (Domain Analysis): 领域分析是一个归纳性的、反馈驱动的过程，它系统地分析某个应用领域，定义其核心挑战和设计规模，以便找到高效的技术解决方案

动态绑定 (Dynamic Binding): 动态绑定是将操作名称（消息）和相应的代码（方法，method）的关联推迟到运行时的一种机制。用于实现面向对象语言中的多态

动态链接库 (DLL) (Dynamically Linked Library DLL) 动态链接库是一种可以由多个进程共享的库。它可以动态地链接到进程地址空间内或者从进程地址空间中卸载，其目的是提高应用的灵活性和可扩展性

末端 (Endpoint): 连接的终结点

事件 (Event): 事件是表示某个活动出现的消息，通常在该消息中携带与这个活动相关的数据

事件处理程序 (Event Handler): 事件处理程序对象包含一系列方法，用于处理属于某个应用的事件

异常安全 (Exception-Safe): 如果一段代码（或者它调用的代码）中抛出的异常不会导致资源泄漏或者系统状态不稳定，我们称这段代码是异常安全的

工厂 (Factory): 工厂是指一个方法或函数。它用于创建所有必备的资源，并把它们组装在一起，以完成对象或组件的实例化和初始化

流控制 (Flow Control): 在一个系统中发送者的能力高于接收者的缓存和计算能力时，我们需要一种通信协议避免溢出，这种协议称为流控制

框架 (Framework): 参见应用框架

全关联 (Full Association): 因特网协议领域确定一个TCP连接的五元组，由协议类型、本地地址和端口号、远程地址和端口号组成

函数 (Function): 函数是一个封闭的子程序，调用者可以向其传入一个或多个参数（也可以不传入任何参数），函数可以向其调用者返回一个值（也可以不返回任何值）。函数通常是独立的，与之相对的“方法”（method）往往属于某个类

功能属性 (Functional Property): 系统功能的某个特定方面, 通常与某个特定的功能需求相关联。功能属性可以对应用的用户是可见的, 比如作为某个功能; 也可以是功能实现的某个方面, 比如作为完成该功能的算法

Future (Future): Future允许客户端在方法调用之后的任何时候取得其结果。Future为被调用的方法存储返回值保留了空间。当客户端需要方法结果的时候, 通过阻塞或者轮询的方式等待输出结果保存到Future

网关 (Gateway): 网关对网络中协作的组件进行解耦合, 使之可以相互交互而避免彼此直接依赖

GUI (GUI): 图形用户界面

句柄 (Handle): 由操作系统内核管理的资源的标识。这些资源通常包括网络连接、打开的文件、计时器、同步对象

硬编码 (Hardwiring): 直接使用数字和字符串而不使用变量的编程方式。这种方式不灵活。有时这种做法也称为“魔术数字”, 因为数字本身对于理解其本意没有任何帮助。另一种形式的硬编码是指代码依赖于具体类型

主机 (Host): 网络上可寻址的计算机

HTTP (HTTP): 超文本传输协议, 建立于TCP之上的简单协议, 客户端使用该协议从Web服务器通过GET请求下载内容

幂等初始化 (Idempotent Initialization): 如果一个对象可以多次重新初始化而不会带来任何有害的副作用, 我们称这种初始化为幂等初始化

惯用法 (Idiom): 惯用法是属于某个特定编程语言或者编程环境的模式。它描述了如何使用给定语言或者环境的特性, 用编码的方式实现特定的行为或者结构。该名词也用于更宽泛的情况, 比如和某个编程语言或者环境相关的通用实践, 不一定特指模式

指示事件 (Indication Event): 指示事件包含从客户端发送到服务提供者的请求信息

继承 (Inheritance): 面向对象语言的一个特性, 允许新的类从已有类中派生。继承定义了实现重用或者子类型关系或者二者兼有。根据语言的不同, 有的语言支持多继承, 有的仅支持单继承

内联 (Inlining): 内联是代码展开的一种方式, 它以直接插入函数或者方法的代码的方式来替换调用该函数或者方法的代码。内联长的方法或者函数会造成“肿胀”, 会在存储和分页上带来负面影响

实例 (Instance): 从具体类中创建的对象。在面向对象环境中经常和“对象”作为同义词使用。这个名词也用于其他的环境 (参见Instantiation)

实例化 (Instantiation): 从一个模板中创建一个新的实例的机制称为实例化。这个术语可以用于多个情形。对象由类实例化而来。C++中的模板可以通过实例化创建新的类或者函数。应用框架可以通过实例化创建一个具体的应用。有时我们使用“实例化某个模式”的说法, 表示在某个特定的应用上下文中采用这个模式, 并补充必要的细节来实现之

调解 (Intercession): 调解是指由一个系统对自身的结构、行为或状态所做的增加或者修改

接口 (Interface): 类、组件、子系统或者应用中可以公开访问的部分。接口这个术语也常用来

指向一个程序构建，它从概念上来说等同于完全抽象的类

因特网 (Internet): 全球性的基于IP的“网之网”。人们普遍认为因特网是继火和MTV之后人类最重要的发明

因特网协议 (IP) (Internet Protocol, IP): 一个网络层协议，用来执行分段 (segmentation)、重组 (reassembly) 和数据包的路由

内联网 (Intranet): 公司或者其他组织内部的计算机网络。这种网络可以阻止外部的不安全访问，从而使用因特网通信技术提供一个公司内部的信息交换、协同工作和工作流平台

进程间通信 (IPC) (Interprocess Communication, IPC): 在不同地址空间中的不同进程之间的通信。IPC机制的例子包括共享内存、UNIX管道、消息队列和套接字通信

内省 (Introspection): 由系统自身完成的对选定的结构、行为或者系统状态的某方面的检查

不变式 (Invariant): 在某个特定的时间或空间点上对象、组件或者模块的状态属性保存不变的属性。例如：“在运行到foo方法的第50行时，总是能够满足 $a < b$ ，这是一个不变式”

抖动 (Jitter): 如果一连串操作的延迟出现偏差，我们称之为抖动。抖动降低了应用的可预测性，因此对于AV流或者实时应用等应用类型是应该尽量避免的

晚期绑定 (Late Binding): 同动态绑定

延迟 (Latency): 操作的推后

层 (Layer): 层是用来定义构成层级结构的一系列服务的抽象概念。第n层是第n-1层的服务的消费者，同时为第n+1层提供服务

负载均衡 (Load Balancing): 负载均衡是一种将来自客户端的负载分发到网络中多个进程和主机的技术

锁 (Lock): 锁是一种用于实现某种类型的临界区的机制。锁可以串行地获得和释放，比如静态互斥件，它可以加入到类中去。如果多个线程同时要获得锁，则只有一个线程能够成功，其他线程将被阻挡在外面直到锁有效。其他的锁机制，比如信号 (semaphore) 或者读写锁，定义了不同的同步语义

封送处理 (Marshaling): 将未封送的 (unmarshaled) 消息从专属于某个主机的形式转换为独立于主机的形式

消息 (Message): 消息用于在对象、线程或者进程之间通信。在面向对象系统中，消息这个词表示选择调用对象的某个操作或者方法。这类消息是同步的，也就意味着发送者必须等待接收者完成调用的操作。线程和进程经常异步通信，消息发送者不等待接受者的回复而是继续执行。远程程序调用 (RPC) 是实现网络中同步IPC的一种方法。然而在IPC通信协议中的消息是由协议定义好的结构组成，并且通常对高层不可见。因此，消息队列系统——比如IMB的MQSeries或微软的MSMQSeries消息——提供了一个用户定义段，高层可以籍此隐式传输用户数据

消息传递 (Message Passing): 线程或进程之间交换消息的一种IPC机制 (对比共享内存)

方法 (Method): 由对象实现的操作。方法属于某个类。该名词也用于“软件开发方法”中，这时候表示一套规则、指南和术语，工程师在软件开发过程中应用这种“方法”

方法闭合 (Method Closure): 方法闭合指的是包含了方法的上下文的对象, 它可以包含方法的参数、对servant或者处理该方法的完成处理程序 (completion handler), 还有可能包括该方法返回结果的Future

中间件 (Middleware): 一系列层和组件, 提供可重用的通用服务和网络编程机制。中间件位于操作系统及其协议栈之上, 而位于特定应用的结构和功能之下

模块 (Module): 软件系统的语义或者概念上的实体, 通常可以认为是组件或者子系统的同义词。有时候模块也表示编译单元或者文件。这里我们使用它的前一个含义。也有人把它和包等同使用, 表示拥有自己的命名空间的代码段

监视器 (Monitor): 监视器将函数和内部变量封装到线程安全的模块中。为了防止竞争状态, 监视器包含一个锁, 同一时刻只允许存在一个活动的线程。想要暂时离开的线程可以阻塞到一个条件变量上

摩尔定律 (Moore's Law): 一个精确得令人吃惊的预言, 它指出微芯片技术的变化速度是微芯片内的元件数量每年翻一番。当1965年Gordon Moore准备一个演讲的时候, 他注意到截至那个时候, 微芯片的容量以每年翻一番的速度增长。由于在过去的几年内, 这种变化的速度略有降低, Gordon Moore于是同意修改这个定义以反映现实的状态, 即每十八个月或者两年翻一番

多播 (Multicast): 多播是一种可以让客户端向多个服务器传输消息的通信协议

多重继承 (Multiple Inheritance): 多重继承是指一个类可以有多于一个父类的继承方式

互斥件 (Mutex): 互斥件是一种“相互排斥”的锁机制, 确保在临界区一次只有一个线程处于活动状态, 从而避免发生竞争状态

网络 (Network): 一种允许主机和其他设备交换消息的通信媒介

网络接口 (Network Interface): 一种连接网络和主机的硬件设备

非功能属性 (Non-functional Property): 系统中与功能无关的特性。非功能属性包括开发属性, 比如可适应性、可扩展性、可维护性, 还有操作属性, 比如易用性、性能、可靠性和可伸缩性 (参见功能属性和操作属性)

对象 (Object): 面向对象系统中可标示的实体。对象通过执行方法 (操作) 来对消息做出响应。对象可以包括数据值和对其他对象的引用, 这合起来定义了对象的状态。所以描述一个对象要包括其状态、行为和标识

面向对象语言 (Object-Oriented Language): 面向对象语言的主要特征是它支持继承、静态和动态的多态, 以及异常处理

ORB (Object Request Broker, ORB): 中间件中的一层。它允许客户端调用分布式对象的方法, 而不必关心对象的位置、编程语言、操作系统平台、通信协议或者硬件

在线协议 (On-the-wire Protocol): 在线协议定义了高层通信中间件 (如DCE、CORBA或者Java RMI) 或其他通信协议 (如HTTP) 是如何将消息或者对象转换成可以“在线 (网络) 上”传输的缓存 (buffer)。“线”这个词包括了各种传输媒介, 比如微波、光纤和音频信号

单向方法调用 (One-way Method Invocation): 单向方法是指调用一个方法, 该方法只向服务器

- 对象传入参数，而不从服务器接收任何返回结果、错误值或者其他信息（对比双向方法调用）
- 操作系统（Operating System）**：一系列服务和API的集合，这些服务和API根据应用和用户的请求管理硬件和软件资源
- 操作系统内核（Operating System Kernel）**：一系列核心操作系统服务的集合，比如进程和线程管理、虚拟内存、进程间通信（IPC）
- 操作属性（Operational Property）**：一种非功能性属性，该属性指出与软件系统的高效使用相关的需求，比如稳定性、性能、可伸缩性、可靠性和安全性。操作性属性直接影响软件系统的可用性和可接受程度。因此，主要是客户和软件系统的用户对运营性属性感兴趣，并受其影响。对比开发性属性
- 带外（Out-of-Band）**：一种协议或者机制，它出现在正常的“带内”（in-band）序列之外；也用来指只有特定的系统或者客户端需要的数据
- 包（Packet）**：一种用于传递TCP/IP协议中头和数据信息的信息
- 并行（Parallelism）**：对象、组件或者系统“物理上同步”执行操作的能力（对比并发）
- 参数（Parameter）**：传递给函数或者方法或者参数化类型的某个数据类型的实例或者对象
- 参数化类型（Parameterized Type）**：一种编程语言的特性，允许一个类可以由其他的类型参数化。Java和C++等多种语言均支持不同类型的参数化机制（对比模板）
- 被动模式套接字（Passive-Mode Socket）**：参见套接字
- 被动连接设施（Passive Connection Establishment）**：如果一个应用被动地接受来自远程应用的连接，我们称之为被动连接设施（对比主动连接设施）
- 被动对象（Passive Object）**：如果一个对象借用其调用者的线程来执行自身的方法，我们称之为被动对象（对比主动对象，Active Object。）
- 模式（Pattern）**：模式用来描述经常出现的设计问题，这些问题出现在某个设计背景之下。模式往往是经过验证的好的解决方案。它描述了方案中的参与者、参与者的职责和关系，以及它们在一起是如何协作的
- 模式组合（Pattern Compound）**：由一族模式组成的模式。一个经常出现的模式群有必要单独定义为一个模式。模式组合也称为“复合模式”或者“组合模式”
- 模式语言（Pattern Language）**：一组相关联的模式定义了系统地解决软件开发问题一个过程，我们称之为模式语言
- 模式序列（Pattern Sequence）**：我们经常需要按照一个序列应用一组模式来为某个特定的状况创建一个特定的架构或者设计，这个序列称为模式序列。从模式语言的角度看，模式序列代表了语言中的一个特定路径
- 模式故事（Pattern Story）**：模式故事讲述了在构造某个具体系统或者某种特定设计时所使用的模式序列以及遇到的设计问题
- 点对点（Peer-to-Peer）**：在分布式系统中，节点就是可以相互通信的进程。与客户端-服务器架构中的组件不同，节点可以作为客户端、服务器，或者两者兼有，甚至可以动态地切换角色
- 平台（Platform）**：平台是指实现系统所需要的硬件和/或软件的组合。软件平台包括操作系统、

库和框架。平台实现了一个虚拟的机器，应用运行于这个虚拟机之上

多态 (Polymorphism): 多态就是说一个名字可能代表不同的东西。例如，一个函数名可能这会儿是这种行为，过一会儿又是另一种行为；一个变量也可能绑定到不同类型的对象上。这个概念使得实现基于抽象的灵活的系统成为可能。在面向对象语言中，多态是通过操作的动态绑定机制实现的。这意味着同一段代码根据其协作的对象的不同可能会有不同的行为

端口 (Port): 通信的一个端点

端口号 (Port Number): 一个16-bit的数字，用于标示TCP协议中的一个通信端点

优先级倒置 (Priority Inversion): 一种调度上的风险，如果低优先级的线程或者请求阻塞了高优先级的线程或者请求的执行，我们称之为优先级倒置

进程 (Process): 进程提供了特定的资源（比如虚拟内存）和保护能力（比如用户/组标识和硬件保护地址空间），这些资源可以被进程内的一个或多个线程使用。与线程相比，进程维护更多的状态信息，在创建、同步和调度上需要更多的开销，进程之间的通信方式往往是消息传递和共享内存

产品族 (Product Family): 参见产品线，Product Line

产品线 (Product Line): 产品线指的是一组产品，它们提供一套共同的服务，满足某个特定市场或者某种使命范围的特定需求。产品线是基于一套相同的核心资源开发出来的。一个产品线中的产品其软件架构和实现大部分是相同的，通常具体的系统都是从同一个框架中派生出来的。当某个产品随着时间的推移而演进的时候，其发布的各个版本也构成一个产品线

产品线架构 (Product-Line Architecture): 产品线架构描述了一组相关联的系统（产品线）的结构属性，通常是描述系统中包含哪些组件，以及这些组件之间的关系。对组件的使用天生就意味着要处理系统中的变化

协议 (Protocol): 一套描述通信各方如何进行信息交互的规则，同时协议还描述了交互信息的语法和语义

协议栈 (Protocol Stack): 一系列具有层级结构的协议组成的协议组

代理 (Proxy): 代理本身可以是组件、服务或者对象，它代表其他的组件、服务或者对象，提供相同的应用接口，同时可以增加一些智能访问控制。例如，远程代理代表一个远程的对象，提供相同的方法接口，但是它本身并不实现请求的行为，而是调用远程的方法来进行实现。而HTTP代理则可以增加安全和缓存功能，以提高系统性能

服务质量 (Quality of Service): 服务质量指的是系统能否提供多种水平的计算和通信能力，比如可用性、带宽、延迟和抖动。这些策略和机制主要是为了控制和提高系统的服务质量

竞争条件 (Race Condition): 竞争条件是一种并发风险，出现在多个线程同时执行同一段关键代码，而没有正确地串行化的情况下

读端临界区 (Read-side Critical Section): 一系列符合下列要求的指令序列：当读端临界区中有一个或多个线程或进程执行的时候，在相应的写端临界区不允许执行任何线程或者进程（对比写端临界区）

多读/单写锁^① (Readers/Writer Lock): 它允许多个线程并发地访问资源, 但是同一时刻只能有一个线程修改资源, 或者干脆不允许同时访问和修改资源

具体化 (Reification): 创建某个抽象的具体实例的动作称为具体化。例如, 一个具体的Reactor实现是Reactor模式的具体化。与之类似, 对象是类的具体化的结果

请求事件 (Request Event): 一个由客户端发送给服务提供者的事件, 要求其根据客户端的要求执行某种处理任务

响应事件 (Response Event): 由服务提供者发送的事件, 其中包含了对客户端请求事件的响应

递归互斥 (Recursive Mutex): 递归互斥是一种锁机制, 拥有互斥对象的线程可以再次获得该互斥对象, 而不会导致自身死锁

重构 (Refactoring): 一种增量性活动, 可以改善组件和框架的内部设计

关系 (Relationship): 组件之间的关联。关系可以是静态的也可以是动态的。静态关系直接写在代码中, 用于处理架构内部组件的放置。动态关系用于处理组件之间的关系, 这种关系可能很难从源代码或者图表中识别出来

远程方法调用 (RMI) (Remote Method Invocation, RMI): 等同于远程过程调用。一台计算机上的客户端通过代理调用一个函数, 执行在服务器计算机上运行的远程对象的函数。这个名词也用于特指Java的RMI机制

远程过程调用 (RPC) (Remote Procedure Call, RPC): 远程过程调用是一种允许运行在客户端计算机上的程序可以执行服务器计算机上的程序, 而不需要程序员显式地用代码写出交互细节的协议

职责 (Responsibility): 对象或者组件在某个上下文中的功能。职责通常表示为一系列语义上相关的操作。职责段是CRC卡的一个元素

角色 (Role): 以相关联元素为上下文的某个设计元素的职责称为角色。例如, 面向对象中的类定义了一个独立的角色, 这个类的所有实例都属于这个角色。另一个例子是接口, 接口的所有实现都属于这个接口所定义的角色。如果一个元素属于某个给定的角色, 它必须为定义这个角色的接口提供一个实现。元素通过实现不同的接口对外表现为不同的角色。不同的元素可以通过实现同一个接口而对外表现为同样的角色, 从而使得客户端可以使用多态的方式对待这些元素, 把它们统一看作该接口的实现。一个实现元素可能同时扮演多种不同的角色, 即使在一个模式中也是如此

调度程序 (Scheduler): 一种用来决定线程或者请求的执行顺序的机制

自描述消息 (Self-Describing Message): 如果一个消息中既包含了描述消息架构的元数据, 又包含了对应于那个架构 (schema) 的值, 我们称这个消息是自描述的消息

信号量 (Semaphore): 信号量是通过计数的方式实现的锁机制。只要计数值大于零, 线程就可以获取这个信号而无需阻塞。当这个计数值变为零之后, 线程便阻塞在这个信号量上, 直到

^① 也可译为“读写锁”, 其含义与“Read/Write Lock”是一致的。——译者注

另一个线程释放这个信号，使得计数值增加并超过零为止

SEP(SEP): 别人的问题(Somebody Else's Problem)、软件工程过程(Software Engineering Process)或者含模式的软件工程(Software Engineering with Patterns)，随便哪个

串行化(Serialization): 串行化是一种防止竞争状态的机制，确保在临界区中一次只执行一个线程。这个名词也用于表示将对象做线性结构的持久化存储，比如字节序列或者XML

Servant (Servant): 由客户端请求触发的组件。当客户端请求到达的时候，Servant自己处理这个请求，或者将这个请求作为子任务委托给其他的组件

服务器(Server): 服务器是指为客户端提供服务的应用，这些服务可能是中间件功能、数据库访问、Web页访问。在分布式对象计算中间件中，这些服务往往用代表分布式对象的Servant实现

服务(Service): 由服务提供者或者服务器提供给它客户端的一系列功能。服务通常由一个或多个组件实现

面向服务的架构(Service-Oriented Architecture): 一种信息系统架构，它可以通过组合松耦合的并可以相互交互的服务来创建应用。这些服务之间基于独立于底层平台和编程语言的接口定义进行通信

共享内存(Shared Memory): 共享内存是一种操作系统机制，它允许计算机的多个进程共享一个内存段(对比消息传递)

单继承(Single Inheritance): 如果一个类最多只能拥有一个直接父类，我们称这种继承方式为单继承

套接字(Socket): 与网络编程相关的一族名词的统称。套接字是通信的一个端点，用于标示某个特定网络地址和端口号。套接字API是大多数操作系统支持的一套函数调用，它用于网络应用建立连接以及套接字端点之间的通信。数据模式套接字用于在通信各方之间交换数据。被动模式套接字用于返回连接的数据状态套接字的句柄

软件架构(Software Architecture): 软件架构描述了软件系统中包含哪些子系统和组件以及它们之间的关系。子系统和组件通常使用不同的视图展示软件系统中相关的功能和非功能属性。系统的软件架构是软件设计活动的结果

饥饿(Starvation): 一种调度风险，如果高优先级的线程持续地抢占优先权从而导致一个或多个低优先级的线程永远不会执行，这种状态就称为饥饿

子类(Subclass): 从父类继承产生的类

子系统Subsystem): 协作执行一个或多个给定服务的一套组件。在软件架构中，子系统被看作是一个独立的实体。它通过与其他子系统和组件交互完成指定的服务

超类(Superclass): 如果其他的类从这个类继承，我们称这个类为超类

同步(Synchronization): 用来协调线程的执行顺序的锁机制

同步I/O(Synchronous I/O): 如果一旦发起I/O操作，调用者阻塞等待操作完成，这种发送和接收数据的机制称为同步I/O

系统(System): 执行一个或多个服务的软件和/或硬件的集合。系统可以是平台，也可以是应用，

或者既是平台也是应用

系统族 (System Family): 参见产品线

模板 (Template): 在C++编程语言中的一个特性, 类和函数可以使用各种类型、常量、函数指针进行参数化。模板常被称为泛型或者参数化类型

线程 (Thread): 线程是一个独立的指令序列, 线程在程序的地址空间内执行, 这个空间可以与其他线程共享。每个线程拥有自己的运行时栈和寄存器, 这使得它可以执行同步I/O操作, 而不需要阻塞其他的线程并发地执行。与进程相比, 线程维护了最低限度的状态信息, 要求相对较少的创建 (spawn)、同步和调度开销, 通常线程通过全局内存中的对象和其他线程通信而不必使用共享内存

连接专属线程 (Thread-per-Connection): 一种并发模型, 将每个网络连接和一个单独的线程关联起来。这个模型在连接的整个生命周期内使用单独的线程处理每个连接到服务器的客户端。它应用于那些必须支持多客户端长周期会话的服务器。对于客户端则没有必要, 比如HTTP1.0Web浏览器, 每个连接只与一个服务请求相关, 这时候thread-per-request模型更为高效

请求专属线程 (Thread-per-Request): 为每一个请求创建 (spawn) 一个新线程的并发模型。它应用于那些必须处理多客户长周期请求事件的服务器, 比如数据库查询。它对于短周期的请求用处不大。如果很多客户端同时发送请求, 它会消耗大量的操作系统资源

线程池 (Thread Pool): 这种并发模型会预先分配多个线程来同时执行请求事件。这个模型是thread-per-request的一个变体, 它通过预先创建一个线程池来分摊线程创建的代价。它应用于想限定所要消耗的操作系统资源数量的服务器。客户请求可以同时执行, 直到并发请求的数量超过池中的线程数目。这时, 后来的请求必须排队, 等到有一个线程可用。然而, 如果在整个执行过程中, 应用实际用到的线程数总是远小于预先分配的线程数, 使用线程池就会浪费线程和操作系统资源

传输控制协议 (TCP) (Transmission Control Protocol): 面向连接的传输协议。它在本地和远端进程之间提供顺序的、不重复的、可靠的数据字节流交换

传输端点 (Transport Endpoint): 在传输层连接每个对等应用 (peer application) 的端点 (endpoint)

传输层 (Transport Layer): 在协议栈中负责端对端数据传输和连接管理的层

传输层接口 (TLI) (Transport Layer Interface): TLI是一套由System V UNIX提供的函数调用, 用于在网络应用中建立连接和在相互连接的传输端点之间通信

双向方法调用 (Two-way Method Invocation): 如果方法调用同时向服务对象传入参数又从服务器获得返回结果, 我们称这个调用是双向方法调用 (对比单向方法调用)

类型安全 (Type-Safety): 编程语言的类型系统强制要求的一个属性, 目的是确保每个类型的实例只能调用有效的操作

Unicode (Unicode): 使用宽字符编码的一种字符编码标准。Unicode包含了绝大多数可书写的字符以及标点符号、数学符号和其他的标示

拆封处理 (Unmarshaling): 参见解封送

上行调用 (Upcall): 从软件架构的底层向高层发起的回调

用户数据报协议 (UDP) (User Datagram Protocol, UDP): 一种用于在本地和远程进程之间交换数据报消息的、不可靠的、非面向连接的传输协议

视图 (View): 视图用于表现软件架构的某个方面, 强调软件系统中的某个特定属性

虚拟机 (Virtual Machine): 为高层应用或者其他的虚拟机提供一系列服务的抽象层

虚拟内存 (Virtual Memory): 一种操作系统机制, 目的是允许开发人员编写的应用使用的地址空间超过计算机上物理内存的大小

写端临界区 (Write-side Critical Section): 一套符合以下要求的指令序列: 最多有一个线程或者进程在写端临界区内执行, 当一个线程或者进程在该写端临界区内执行的时候, 在相应的读端临界区中不可以有任何线程或者进程在执行 (对比读端临界区)



参 考 书 目

- [ACM01] D. Alur, J. Crupi, D. Malks: *Core J2EE Patterns, Best Practices and Design Strategies*, Second edition, Prentice Hall, 2005
- [AIS77] C. Alexander, S. Ishikawa, M. Silverstein with M. Jacobson, I. FiksdahlKing, S. Angel: *A Pattern Language—Towns · Buildings · Construction*, Oxford University Press, 1977
- [Ale79] C. Alexander: *The Timeless Way of Building*, Oxford University Press, 1979
- [Ale01] A. Alexandrescu: *Modern C++ Design: Generic Programming and Design Patterns Applied*, Addison-Wesley 2001
- [And96] B. Anderson: *Null Object*, presented at the First European Conference on Pattern Languages of Programming, EuroPLOP 1996, 1996
- [Apache06] Apache Software Foundation: *Web Service Invocation Framework*, <http://ws.apache.org/wsif/>
- [BEA06] BEA Systems: *BEA MessageQ Product Overview*, <http://www.bea.com/framework.jsp?CNT=overview.htm&FP=/content/products/more/messageq/>, BEA Systems, 2006
- [Beck97] K. Beck: *Smalltalk Best Practices*, Prentice Hall, 1996
- [BeCu87] K. Beck, W. Cunningham: *Using Pattern Languages for Object-Oriented Programs*, submission to the OOPSLA '87 workshop on Specification and Design for Object-Oriented Programming, October 1987
- [BeLe76] L. A. Belady, M. M. Lehman: *A Model of Large Program Development*, IBM Systems Journal, Volume 15(3), pp. 225–252, 1976
- [Bell06] A. E. Bell: *Software Development Amidst the Whiz of Silver Bullets*. . . , ACM Queue Volume 4, No. 5, June 2006
- [Ben86] J. Bentley: *Little Languages*, Communications of the ACM, 29(8), pp. 711–721, August 1986
- [BGB00] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, D. Hardin, M. Turnbull: *The Real-Time Specification for Java*, Addison-Wesley, 2000
- [Bir05] K. Birman: *Reliable Distributed Systems: Technologies, Web Services, and Applications*, Springer, 2005
- [Bosch00] J. Bosch: *Design and Use of Software Architectures—Adapting and Evolving a Product-Line Approach*, Addison-Wesley, 2000
- [Box97] D. Box: *Essential COM*, Addison-Wesley, 1997
- [Bus03] F. Buschmann: *Notes on The Forgotten Art of Building Good Software Architectures*, Tutorial at the Eighth Conference on Java and Object-Oriented Technology, JAOO 2003, Aarhus, Denmark, 2003

- [BW95] K. Brown, B. Whitenack: *Crossing Chasms: A Pattern Language for ObjectRDBMS Integration*, in [PLoPD2], 1995
- [Celtix06] Celtix Enterprise Service Bus: *User Guides*, <http://celtix.objectweb.org/>, 2006
- [CLF93] D. de Champeaux, D. Lea, P. Faure: *Object-Oriented System Development*, Addison-Wesley, 1993
- [CINo01] P. Clements, L. Northrop: *Software Product Lines: Practices and Patterns*, Addison-Wesley, 2001
- [CMH83] K. M. Chandy, J. Misra, and L. M. Haas: *Distributed Deadlock Detection*, ACM Transactions on Computer Systems, 1(2), 143–156, May 1983.
- [Cope92] J. O. Coplien: *Advanced C++ Programming Styles and Idioms*, Addison-Wesley, 1991
- [Cope96] J. O. Coplien: *Software Patterns*, SIGS Books, New York, New York, 1996. See also <http://users.rcn.com/jcoplien/Patterns/WhitePaper/>.
- [Cope98] J. O. Coplien: *Multi-Paradigm Design for C++*, Addison-Wesley, 1998
- [CSKO+02] A. Corsaro, D. C. Schmidt, R. Klefstad, Carlos O’Ryan: *Virtual Component: A Design Pattern for Memory-Constrained Embedded Applications*, Proceedings of the Ninth Annual Conference on the Pattern Languages of Programs, Monticello, Illinois, September 2002
- [CzEi02] C. Czarnecki, U. Eisenecker: *Generative Programming, Methods, Tools and Applications*, Addison-Wesley, 2000
- [DBOSG05] G. Deng, J. Balasubramanian, W. Otte, D. C. Schmidt, A. Gokhale: *DAnCE: A QoS-Enabled Component Deployment and Configuration Engine*, Proceedings of the Third Working Conference on Component Deployment, Grenoble, France, November, 2005
- [DeGe04] J. Dean, S. Ghemawat: *MapReduce: Simplified Data Processing on Large Clusters*, OSDI ’04—Sixth Symposium on Operating System Design and Implementation, San Francisco, CA, December, 2004.
- [DOC] The DOC group open-source software, Institute for Software Integrated Systems, Vanderbilt University, www.dre.vanderbilt.edu
- [DWT04] A. Dennis, B. Haley Wixom, D. Tegarden: *Systems Analysis and Design with UML Version 2.0: An Object-Oriented Approach*, John Wiley & Sons, 2004
- [DyAn98] P. Dyson, B. Anderson: *State Patterns*, in [PLoPD3], 1997
- [Eng99] J. Engel: *Programming for the Java Virtual Machine*, Addison-Wesley, 1999
- [EPL02] R. Elfwing, U. Paulsson, L. Lundberg: *Performance of SOAP in Web Service Environment Compared to CORBA*, Proceedings of the Ninth Asia-Pacific Software Engineering Conference, December 2002, Gold Coast, Australia
- [Evans03] E. Evans: *Domain-Driven Design*, Addison-Wesley, 2003
- [FBBOR99] M. Fowler, K. Beck, J. Brant, W. Opdyke, D. Roberts: *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999
- [FeMac02] A. Ferrara, M. MacDonald: *Programming .NET Web Services*, O’Reilly, 2002
- [FGMFB97] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, T. Berners-Lee: *Hypertext Transfer Protocol—HTTP/1.1*, Network Working Group, RFC 2068, January 1997
- [FJS99a] M. Fayad, R. Johnson, D. C. Schmidt (eds.): *Implementing Application Frameworks: Object-Oriented Frameworks at Work*, John Wiley & Sons, New York, NY, 1999
- [FJS99b] M. Fayad, R. Johnson, D. C. Schmidt (eds.): *Building Application Frameworks: Object-Oriented Foundations of Framework Design*, John Wiley & Sons, New York, NY, 1999

- [Fow97] M. Fowler: *Analysis Patterns*, Addison-Wesley, 1997
- [Fow03a] M. Fowler: *Patterns of Enterprise Application Architecture*, Addison-Wesley, 2002
- [Fow03b] M. Fowler: *UML Distilled: A Brief Guide to the Standard Object Modeling Language*, Third edition, Addison-Wesley, 2003
- [Fow06] M. Fowler: *The Model-View-Presenter Pattern*, <http://www.martinfowler.com/eaDev/ModelViewPresenter.html>, 2006
- [FoYo99] B. Foote, J. Yoder: *Big Ball of Mud*, in [PLoPD4], 1999
- [Fri06] T. L. Friedman: *The World is Flat: A Brief History of the Twenty-First Century*, expanded and updated version, Farrar, Straus and Giroux, 2006
- [Gar05] J. J. Garrett: *Ajax: A New Approach to Web Applications*, February 2005, <http://adaptivepath.com/publications/essays/archives/000385.php>
- [Gam92] E. Gamma: *Objektorientierte Software-Entwicklung am Beispiel von ET++: Design-Muster, Klassenbibliotheken, Werkzeuge*, Springer, 1992
- [Gam97] E. Gamma: personal communication, 1997
- [GoF95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995
- [GS97a] A. Gokhale, D. C. Schmidt: *Design Principles and Optimizations for HighPerformance ORBs*, OOPSLA '97 Poster Session, Atlanta, GA, ACM, 1997
- [GS97b] A. Gokhale, D. C. Schmidt: *Evaluating the Performance of Demultiplexing Strategies for Real-Time CORBA*, Proceedings of GLOBECOM '97, Phoenix, AZ, IEEE, 1997
- [GS98] A. Gokhale, D. C. Schmidt: *Optimizing A CORBA IIOP Protocol Engine for Minimal Footprint Multimedia Systems*, submitted to *IEEE Journal on Selected Areas in Communications*, special issue on Service Enabling Platforms for Networked Multimedia Systems, 1998
- [HBS+02] M. Hapner, R. Burrige, R. Sharma, J. Fialli, K. Haase: *Java Message Service API Tutorial and Reference: Messaging for the J2EE Platform*, Addison-Wesley, 2002
- [Hearsay02] Kloster Hearsay (the daily EuroPLoP newspaper), Issue 02/2002, Joe Bergin: *Do the Right Thing*, Irsee, Germany, 2002
- [Hen99] K. Henney: *Collections for States*, Proceedings of the Fourth European Conference on Pattern Languages of Programming, EuroPLoP 1999, Irsee, Universitätsverlag Konstanz, July 2001
- [Hen00a] K. Henney: *Patterns of Value*, *Java Report* 5(2), February 2000
- [Hen00b] K. Henney: *Value Added*, *Java Report* 5(4), April 2000
- [Hen00c] K. Henney: *A Tale of Two Patterns*, *Java Report*, SIGS Publications, December 2000
- [Hen01a] K. Henney: *C++ Patterns—Executing Around Sequences*, Proceedings of the Fifth European Conference on Pattern Languages of Programming, EuroPLoP 2000, Irsee, Universitätsverlag Konstanz, July 2001
- [Hen01b] K. Henney: *C++ Patterns—Reference Accounting*, Proceedings of the Sixth European Conference on Pattern Languages of Programming, EuroPLoP 2001, Irsee, Universitätsverlag Konstanz, July 2002
- [Hen01c] K. Henney: *A Tale of Three Patterns*, *Java Report*, SIGS Publications, October 2001
- [Hen02a] K. Henney: *Null Object*, Proceedings of the Seventh European Conference on Pattern Languages of Programming, EuroPLoP 2002, Irsee, Universitätsverlag Konstanz, July 2003

- [Hen02b] K. Henney: *Patterns in Java: The Importance of Symmetry*, JavaSpektrum, Issue 6, 2002, SIGS-DATACOM GmbH, Germany
- [Hen02c] K. Henney: *Methods for States*, Proceedings of the First Nordic Conference on Pattern Languages of Programming, VikingPLoP 2002, Helsingør, Denmark Universitätsverlag Konstanz, July 2003
- [Hen05] K. Henney: *Context Encapsulation—Three Stories, A Language, and Some Sequences*, Proceedings of the Tenth European Conference on Pattern Languages of Programming, EuroPLoP 2005, Irsee, Universitätsverlag Konstanz, July 2006
- [HMS97] J. Hu, S. Mungee, D. C. Schmidt: *Principles for Developing and Measuring High-Performance Web Servers over ATM*, Proceedings of INFOCOM '98, March/April 1998
- [HoWo03] G. Hohpe, B. Woolf: *Enterprise Integration Patterns—Designing, Building, and Deploying Messaging Solutions*, Addison-Wesley, 2003
- [HPS97] J. Hu, I. Pyrali, D. C. Schmidt: *Measuring the Impact of Event Dispatching and Concurrency Models on Web Server Performance Over High-Speed Networks*, Proceedings of the 2nd Global Internet Conference, IEEE, 1997
- [HV99] M. Henning, S. Vinoski: *Advanced CORBA Programming with C++*, AddisonWesley, 1999
- [IBM99] IBM Corporation: *MQSeries Version 5.1 Administration and Programming Examples*, IBM Redbooks, 1999
- [IBM06] IBM Corporation: *Cell Broadband Engine Programming Handbook*, pp. 603, April 2006
- [IEEE96] IEEE: *Threads Extension for Portable Operating Systems*, (Draft 10), February 1996
- [John97] R. Johnson: *Frameworks = Patterns + Components*, Communications of the ACM, M. Fayad, D.C. Schmidt (eds.), Volume 40, No. 10, October 1997
- [Kaye03] D. Kaye: *Loosely Coupled, The Missing Pieces of Web Services*, Rds Associates, 2003
- [KC97] W. Keller, J. Coldewey: *Accessing Relational Databases*, in [PLoPD3], 1997
- [Kel04] A. Kelly: *Encapsulated Context*, Proceedings of the Eighth European Conference on Pattern Languages of Programming, EuroPLoP 2003, Irsee, Universitätsverlag Konstanz, July 2004
- [Kel99] W. Keller: *Object/Relational Access Layer*, in Proceedings of the Third European Conference on Pattern Languages of Programming, EuroPLoP 1998, Irsee, Universitätsverlag Konstanz, July 1999
- [KGS+05] A. S. Krishna, A. Gokhale, D. C. Schmidt, V. P. Ranganath, J. Hatcliff: *Model-Driven Middleware Specialization Techniques for Software Product-Line Architectures in Distributed Real-Time and Embedded Systems*, Proceedings of the MODELS 2005 workshop on MDD for Software Product-Lines, Half Moon Bay, Jamaica, October 2005
- [KLLM95] G. Kiczales, R. DeLine, A. Lee, C. Maeda: *Open Implementation—Analysis and DesignTM of Substrate Software*, Tutorial #21 of OOPSLA '95, October 1995
- [Kof04] T. Kofler: *Robust Iterators for ET++*, *Structured Programming*, Volume 14, Number 2, pp. 62–85, 1993
- [KSK04] A. S. Krishna, D. C. Schmidt, R. Klefstad: *Enhancing Real-Time CORBA via Real-Time Java Features*, Proceedings of the Twenty-Fourth IEEE International Conference on Distributed Computing Systems (ICDCS), Tokyo, Japan, May 2004
- [KSS05] A. Krishna, D. C. Schmidt, M. Stal: *Context Object: A Design Pattern for Efficient Middleware Request Processing*, Proceedings of the Twelfth Pattern Language of Programming Conference,

- Allerton Park, Illinois, September 2005
- [Lak95] J. Lakos: *Large-Scale C++ Software Design*, Addison-Wesley, 1995
- [Lea02] D. Lea: personal communication, May 2002
- [Lea99] D. Lea: *Concurrent Programming in Java: Design Principles and Patterns*, Second edition, Addison-Wesley, 2000
- [Lee06] E. A. Lee: *The Problem with Threads*, IEEE Computer, May 2006
- [Lew95] B. Lewis, D. J Berg: *Threads Primer: A Guide to Multithreaded Programming*, Prentice Hall, 1995
- [LGS00] D. L. Levine, C. D. Gill, D. C. Schmidt: *Object Lifetime Manager—A Complementary Pattern for Controlling Object Creation and Destruction*, in *Design Patterns in Communications*, L. Rising (ed.), Cambridge University Press, 2001
- [Lin03] D. Linthicum: *Next Generation Application Integration: From Simple Information to Web Services*, Addison-Wesley, 2003
- [LY99] T. Lindholm, F. Yellin: *The Java Virtual Machine Specification*, Second edition, Addison-Wesley, 1999
- [Maf96] S. Maffeis: *The Object Group Design Pattern*, Proceedings of the 1996 USENIX Conference on Object-Oriented Technologies, USENIX, Toronto, Canada, June 1996
- [MaHa99] V. Matena, M Hapner: *Enterprise JavaBeans*, Version 1.1, Sun Microsystems Inc., 1999
- [Mar04] R. Martin: *The Dependency Inversion Principle*, C++ Report, Volume 8, No 6, May 1996
- [McK96] P. E. McKenney: *Selecting Locking Designs for Parallel Programs*, in [PLoPD2], 1996
- [MeAl04a] S. Meyers, A. Alexandrescu: *C++ and the Perils of Double-Checked Locking: Part I*, Dr. Dobb's Journal, June 2004
- [MeAl04b] S. Meyers, A. Alexandrescu: *C++ and the Perils of Double-Checked Locking: Part II*, Dr. Dobb's Journal, June 2004
- [Mes95] G. Meszaros: *Half-Object plus Protocol*, in [PLoPD1], 1995
- [Mes96] G. Meszaros: *A Pattern Language for Improving the Capacity of Reactive Systems*, in [PLoPD2], 1996
- [Mey97] B. Meyer: *Object-Oriented Software Construction*, Second edition, Prentice Hall, 1997
- [MMW06] C. McMurtry, M. Mercuri, N. Watling: *Microsoft Windows Communication Foundation: Hands-On!*, Sams, 2006
- [MPY+04] A. Memon, A. Porter, C. Yilmaz, A. Nagarajan, D. C. Schmidt, B. Natarajan: *Skoll: Distributed Continuous Quality Assurance*, Proceedings of the 26th IEEE/ACM International Conference on Software Engineering, Edinburgh, Scotland, May 2004
- [MS03] Microsoft Corporation: *Enterprise Solution Patterns Using Microsoft .NET Version 2.0*, Microsoft Press, 2003
- [MSS00] S. Mungee, N. Surendran, D. C. Schmidt: *The Design and Performance of a CORBA Audio/Video Streaming Service*, in *Design and Management of Multimedia Information Systems: Opportunities and Challenges*, M. Syed (ed.), Idea Group Publishing, Hershey, PA, 2000
- [Mule06] Mule Enterprise Service Bus: user documentation, <http://mule.codehaus.org/>, 2006
- [OASIS06a] Organization for the Advancement of Structured Information Standards: *Reference Model for Service-Oriented Architecture*, Version 1.0, Committee Specification, July 2006,

- <http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf>
 Organization for the Advancement of Structured Information Standards: *Web Services Base Notification*, Version 1.3, Committee Specification, July 2006
- [OG94] The Open Group: *DCE: Remote Procedure Call*, available at <http://www.opengroup.org/bookstore/catalog/c309.htm>, 1994
- [OKS+00] C. O’Ryan, F. Kuhns, D. C. Schmidt, O. Othman, J. Parsons: *The Design and Performance of a Pluggable Protocols Framework for Real-Time Distributed Object Computing Middleware*, Proceedings of the ACM/IFIP Middleware 2000 Conference, Pallisades, New York, April 2000
- [OMG02] ObjectManagement Group: *CORBA Component Model*, Version 3.0, June 2002
- [OMG03a] Object Management Group: *Real-Time CORBA Specification (static scheduling)*, Version 1.2, January 2005 <http://www.omg.org/cgi-bin/doc?formal/05-01-04>
- [OMG03b] Object Management Group: *Specification for Deployment and Configuration of Component-based Distributed Applications*, adopted submission, OMG document ptc/03-07-08, 2003
- [OMG04a] Object Management Group: *Common Object Request Broker Architecture*, Version 3.0.3, March 2004
- [OMG04b] Object Management Group: *Lightweight CORBA Component Model*, draft adopted specification, May 2004 <http://www.omg.org/cgi-bin/doc?realtime/2003-05-05>
- [OMG04c] Object Management Group: *Notification Service Specification*, Version 1.1, October 2004
- [OMG05a] ObjectManagement Group: *Real-Time CORBA Specification (dynamic scheduling)*, Version 1.2, January 2005
- [OMG05b] Object Management Group: *Real-Time Data Distribution Service*, Version 1.1, December 2005
- [Pal05] D. Pallmann: *Programming INDIGO*, Mierosoft Press, 2005
- [Par94] D. L. Parnas: *Software Aging*, Proceedings of the Sixteenth International Conference on Software Engineering (ICSE-16), Sorrento, Italy, May 1994
- [PHS96] I. Pyarali, T. H. Harrison, D. C. Schmidt: *Design and Performance of an Object-Oriented Framework for High-Performance Electronic Medical Imaging*, *USENIX Computing Systems*, Volume 9, November/December 1996
- [PLoPD1] J. O. Coplien, D. C. Schmidt (eds.): *Pattern Languages of Program Design*, Addison-Wesley, 1995 (a book publishing the reviewed Proceedings of the First International Conference on Pattern Languages of Programming, Monticello, Illinois, 1994)
- [PLoPD2] J. O. Coplien, N. Kerth, J. Vlissides (eds.): *Pattern Languages of Program Design 2*, Addison-Wesley, 1996 (a book publishing the reviewed Proceedings of the Second International Conference on Pattern Languages of Programming, Monticello, Illinois, 1995)
- [PLoPD3] F. Buschmann, R. C. Martin, D. Riehle (eds.): *Pattern Languages of Program Design 3*, Addison-Wesley, 1997 (a book publishing selected papers from the Third International Conference on Pattern Languages of Programming, Monticello, Illinois, USA, 1996, the First European Conference on Pattern Languages of Programming, Irsee, Bavaria, Germany, 1996, and the Telecommunication Pattern Workshop at OOPSLA ’96, San Jose, California, USA, 1996)
- [PLoPD4] B. Foote, N. B. Harrison, H. Rohnert (eds.): *Pattern Languages of Program Design 4*, Addison-Wesley, 1999 (a book publishing selected papers from the Fourth and Fifth International

- Conference on Pattern Languages of Programming, Monticello, Illinois, USA, 1997 and 1998, and the Second and Third European Conference on Pattern Languages of Programming, Irsee, Bavaria, Germany, 1997 and 1998)
- [PLoPD5] D. Manolescu, J. Noble, M. Völter (eds.): *Pattern Languages of Program Design 5*, Addison-Wesley, 2006 (a book publishing selected papers from the Pattern Languages of Programming conference series from 1999–2004)
- [POSA1] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal: *Pattern-Oriented Software Architecture, Volume 1: A System of Patterns*, John Wiley & Sons, 1996
- [POSA2] D. C. Schmidt, M. Stal, H. Rohnert, F. Buschmann: *Pattern-Oriented Software Architecture, Volume 2: Patterns for Concurrent and Networked Objects*, John Wiley & Sons, 2000
- [POSA3] P. Jain, M. Kircher: *Pattern-Oriented Software Architecture, Volume 3: Patterns for Resource Management*, John Wiley & Sons, 2004
- [POSA5] F. Buschmann, K. Henney, D. C. Schmidt: *Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages*, John Wiley & Sons, 2007
- [POSIX95] *Information Technology—Portable Operating System Interface (POSIX)—Part 1: System Application: Program Interface (API) [C Language]*, 1995
- [PP03] M. Poppendieck, T. Poppendieck: *Lean Software Development: An Agile Toolkit for Software Development Managers*, Addison-Wesley, 2003
- [PPR] *The Portland Pattern Repository*, <http://www.c2.com>
- [Pree94] W. Pree: *Design Patterns for Object-Oriented Software Development*, Addison-Wesley, 1994
- [PRS+00] I. Pyarali, C. O’Ryan, D. C. Schmidt, N. Wang, V. Kachroo, A. Gokhale: *Using Principle Patterns to Optimize Real-Time ORBs*, IEEE Concurrency Magazine, Volume 8, Number 1, January/March 2000
- [PSC+01] I. Pyarali, M. Spivak, R. K. Cytron, D. C. Schmidt: *Optimizing Threadpool Strategies for Real-Time CORBA*, Proceedings of the ACM Workshop on Optimization of Middleware and Distributed Systems, pp. 214–222, June, 2001, Snowbird, Utah
- [Rago93] S. Rago: *UNIX System V Network Programming*, Addison-Wesley, 1993
- [Ram02] I. Rammer: *Advanced .NET Remoting*, APress, 2002
- [Ris01] L. Rising: *Design Patterns in Communications Software*, Cambridge University Press, 2001
- [RKF92] W. Rosenberry, D. Kenney, G. Fischer: *Understanding DCE*, O’Reilly and Associates, Inc. 1992
- [SC99] D. C. Schmidt, C. Cleeland: *Applying Patterns to Develop Extensible ORB Middleware*, IEEE Communications Magazine, special issue on Design Patterns, April 1999
- [SCA05] *Service Component Architecture: Assembly Model Specification*, Version 0.9, November 2005
- [Sch00] D. C. Schmidt: *Applying a Pattern Language to Develop Application-Level Gateways*, in *Design Patterns in Communications*, ed. Linda Rising, Cambridge University Press, 2000
- [ScSc01] R. E. Schantz, D. C. Schmidt: *Middleware for Distributed Systems: Evolving the Common Structure for Network-Centric Applications*, in *Encyclopedia of Software Engineering*, J. Marciniak, G. Telecki (eds.), John Wiley & Sons, New York, 2001
- [ScVi99] D. C. Schmidt, S. Vinoski: *Collocation Optimizations for CORBA*, C++ Report, SIGS, Volume 11, Number 10, pp. 47–52, November/December 1999
- [SDL05] A. Prinz, J. Reed, R. Reed (eds.): *SDL 2005: Model Driven*, Proceedings of the 12th International

- SDL Forum, Grimstad, Norway, June 20–23, 2005, Springer, 2005
- [SFHBS06] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, P. Sommerlad: *Security Patterns: Integrating Security and Systems Engineering*, John Wiley & Sons, 2006
- [SGS01] V. Subramonian, C. Gill, D. Sharp: *Towards a Pattern Language for Networked Embedded Software Technology Middleware*, ACM OOPSLA Workshop on Towards Patterns and Pattern Languages for OO Distributed RealTime and Embedded Systems, Tampa Bay, Florida, October 2001
- [SH02] D. C. Schmidt, S. D. Huston: *C++ Network Programming, Volume 1: Mastering Complexity with ACE and Patterns*, Addison-Wesley, 2002
- [SH03] D. C. Schmidt, S. D. Huston: *C++ Network Programming, Volume 2: Systematic Reuse with ACE and Frameworks*, Addison-Wesley, 2003
- [SMFG00] D. C. Schmidt, S. Mungee, S. Flores-Gaitan, A. Gokhale: *Software Architectures for Reducing Priority Inversion and Non-Determinism in Real-Time Object Request Brokers*, Journal of Real-Time Systems, special issue on Real-Time Computing in the Age of the Web and the Internet, ed. A. Stoyen, Kluwer, 2000
- [SN96] R. W. Schulte, Y. V. Natis: *Service Oriented Architectures*, Part 1, SSA Research Note SPA-401-068, Gartner, 12 April 1996
- [SNG+02] D. C. Schmidt, B. Natarajan, A. Gokhale, N. Wang, C. Gill, *TAO: A PatternOriented Object Request Broker for Distributed Real-Time and Embedded Systems*, *IEEE Distributed Systems Online*, Volume 3, Number 2, February, 2002
- [Sol98] D. A. Solomon: *Inside Windows NT*, Second edition, Microsoft Press, 1998
- [Som97] P. Sommerlad: *Manager*, in [PLoPD3], 1997
- [Ste93] W. R. Stevens: *TCP/IP Illustrated, Volume 1*, Addison-Wesley, 1993
- [Ste98] W. R. Stevens: *Unix Network Programming, Volume 1: Networking APIs: Sockets and XTI*, Second edition, Prentice Hall, 1998
- [Str97] B. Stroustrup: *The C++ Programming Language*, Third edition, AddisonWesley 1997
- [StRa05] W. R. Stevens, S. A. Rago: *Advanced Programming in the UNIX environment*, Second edition, Addison-Wesley, 2005
- [StSc05] M. Stal, D. C. Schmidt: *Activator*, Proceedings of the Twelfth Pattern Language of Programming Conference, Allerton Park, Illinois, September 2005
- [Sun88] Sun Microsystems: *Remote Procedure Call Protocol Specification*, Sun Microsystems Inc., RFC-1057, June 1988
- [Sun03] Sun Microsystems: *Enterprise JavaBeans Specification, Version 2.1*, Sun Microsystems Inc., November 2003
- [Sun04a] Sun Microsystems: *Enterprise JavaBeans Specification, Version 3.0, early draft*, Sun Microsystems Inc., June 2004
- [Sun04b] Sun Microsystems: *Java Message Service (JMS), Version 3.0, early draft*, Sun Microsystems Inc., June 2004
- [Sun04c] Sun Microsystems: *Java Remote Method Invocations (RMI)*, Sun Microsystems Inc., 2004
- [Sut05a] H. Sutter: *The Free Lunch Is Over: A Fundamental Turn Toward Concurrency in Software*, *Dr. Dobbs's Journal*, 30(3), March 2005

- [Sut05b] H. Sutter: *Software and the Concurrency Revolution*, InStat Fall Processor Forum, October 2005
- [Szy02] C. Szyperski: *Component Software: Beyond Object-Oriented Programming*, Second edition, Addison-Wesley, 2002
- [Tan92] A. S. Tanenbaum: *Modern Operating Systems*, Prentice Hall, 1992
- [Tan95] A. S. Tanenbaum: *Distributed Operating Systems*, Prentice Hall, 1995
- [TaSte02] A. S. Tanenbaum, M. van Steen: *Distributed Systems: Principles and Paradigms*, First edition, Prentice Hall, 2002
- [Thai99] T. L. Thai: *Learning DCOM*, O'Reilly, 1999
- [Vin03] S. Vinoski: *Toward Integration: Integration with Web Services*, *IEEE Internet Computing*, November/ December 2003, pp. 75–77, 2003
- [Vin04a] S. Vinoski: *An Overview of Middleware*, Ninth International Conference on Reliable Software Technologies Ada-Europe 2004, Palma de Mallorca, 14–18 June 2004
- [Vin04b] S. Vinoski: *WS-Nonexistent Standards*, *IEEE Internet Computing*, November/December 2004, IEEE, 2004
- [Vlis98a] J. Vlissides: *Pattern Hatching: Design Patterns Applied*, Addison-Wesley, 1998
- [Vlis98b] J. Vlissides: *Pluggable Factory*, Part I, C++ Report, November/December 1998
- [Vlis99] J. Vlissides: *Pluggable Factory*, Part II, C++ Report, February 1999
- [VKZ04] M. Völter, M. Kircher, U. Zdun: *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*, John Wiley & Sons, 2004
- [VSW02] M. Völter, A. Schmid, E. Wolff: *Server Component Patterns—Component Infrastructures Illustrated with EJB*, John Wiley & Sons, 2002
- [W3C03] World Wide Web Consortium: *SOAP Version 1.2*, June 2003
- [W3C06a] World Wide Web Consortium: *Web Services Description Language (WSDL) Version 2.0*, June 2006
- [W3C06b] WorldWideWeb Consortium: *Extensible Markup Language (XML) 1.1*, September 2006
- [WK01] J. Weigmann, G. Kilian: *Decentralization with PROFIBUS-DP: Architecture and Fundamentals, Configuration and Use with Step 7*, John Wiley & Sons, 2001
- [Woolf97] B. Woolf: *Null Object*, in [PLoPD3], 1997
- [WRW96] A. Wollrath, R. Riggs, J. Waldo: *A Distributed Object Model for the Java System*, *USENIX Computing Systems*, ed. Douglas C. Schmidt, Volume 9, Number 4, MIT Press, November/December 1996
- [WSG+03] N. Wang, D. C. Schmidt, A. Gokhale, C. Rodrigues, B. Natarajan, J. P. Loyall, R. E. Schantz, C. D. Gill: *QoS-Enabled Middleware*, in *Middleware for Communications*, pp. 131–162, John Wiley and Sons, New York, 2003
- [WWWK96] S. C. Kendall, J. Waldo, A. Wollrath, G. Wyant (eds.): *A Note On Distributed Computing, Technical Reports and Essay Series*, Sun Microsystems Inc., 1996, <http://research.sun.com/techrep/1994/abstract-29.html>

译 后 记

翻译很容易让人“只见树木不见森林”，至少对于我是这样的。而对于这样一本旨在介绍模式语言之洋洋大观的经典图书，只见树木恐怕是远远不够的，所以，我每次不得不先通读一章然后再翻译，校对之前也是先通读一章再校对。

模式也一样，模式看多了也容易“只见树木不见森林”，所以需要一本书帮我们整理一下思路——本书就是这样一本书了。如果把模式比作语言中的单词，把模式序列比作例句，本书就是一本语法书，它将告诉你怎样使用“分布式计算模式语言”中的“单词”。书中通过一个典型的案例（仓库管理流程控制系统）全面介绍了设计和开发分布式系统中可能遇到的各种问题、问题的背景及其解决方案。

有的书是有一说一，而本书则是“弱水三千，取一瓢饮”。书中所介绍的分布式计算模式语言包含了114个模式，所以作者只能将注意力放到讲述模式的本质和模式的关系上面。好在，本书介绍到的绝大部分模式都可以在网络上找到详细的描述。译者也在翻译过程中做了专门的收集，我将在第五卷翻译完之后慢慢整理发布出来。请关注我的博客：<http://designpatterns.cnblogs.com>。

我非常荣幸能够得到这本书的翻译机会。感谢图灵公司对我的信任，特别感谢傅志红、刘江、陈兴璐、朱巍等编辑的支持和帮助。老赵（赵劼）在论坛上的一句玩笑话——“把POSA做好就功德无量了”——令我压力倍增。整个翻译过程中我都是如履薄冰。幸得作者及时回答我的问题，周边朋友的帮忙，才能够把这样一本译稿呈现在读者面前。图灵俱乐部论坛的很多兄弟，不要说谋面，很多连姓名都叫不上来，这里就不一一感谢了。特别感谢李锐、李松峰、be_flying、Telethink、郑柯、贾国伟等同学的大力帮忙。当然还要特别感谢我的主要合译者陈立，他深厚的技术和翻译功底令人钦佩。

我的东家ThoughtWorks给了我非常宽松的环境，使我在没有项目的时候可以占用一些上班时间去翻译，并且在ThoughtWorks有很多同事跟我讨论本书中的翻译问题和技术问题。感谢韩锴、荣浩、李剑等同事的帮助。

感谢我的家人在本书翻译过程中给我的支持和理解，一年多来即使回到家也没有多少时间陪着父母。感谢我的岳父和岳母精心照顾我的生活，感谢我的妻子帮我打字，帮我找到最合适的“中国话”，感谢她对这个“枯燥老公”的容忍。

本书第1~5章、第9章、第11章、第12章、第16章、第19章、第20章由陈立翻译，第10章由韩锴翻译，第17章由李锐翻译，其余部分由肖鹏翻译，全书由肖鹏通稿。由于译者水平有限，翻译错误或者风格不合口味在所难免，欢迎得到您的指正。您的任何意见和建议都可以发表到我的博客，也可以直接给我发邮件：eagle.xiao@gmail.com。

感谢您选择我的译文，期待您的反馈！

肖 鹏

2010年5月于北京



① ② ③ ④ ⑤ ⑥ ⑦
读 者 积 分 赠 书 卡

手机号码: _____ (此为会员编号)

姓 名: _____ 性别: ☐男 ☐女 出生年月: ____年__月

通信地址: _____

邮政编码: _____

电子邮件: _____

您购买的图书是: 22773 / 《面向模式的软件架构 卷4: 分布式计算的
模式语言》 (69.00元)

您获得的会员积分是: 6.9 分

欢迎参加“**有奖DEBUG**”活动。提交本书勘误, 每确认一处即可获赠积分5分。详情见图灵网站。

请沿虚线剪下此页, 寄回图灵公司, 即可成为图灵读者俱乐部的一员 (复印无效)。**积分累计, 可获赠书** (赠书清单见图灵网站)。

邮政编码: 100107

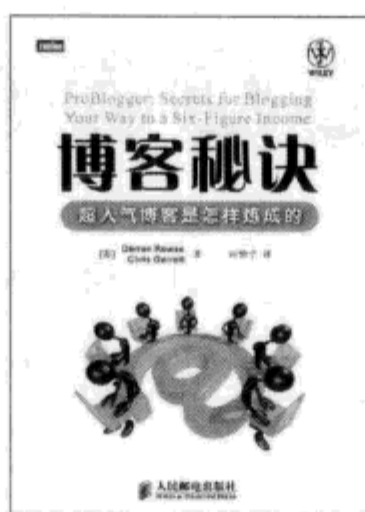
通信地址: 北京市朝阳区北苑路 13 号院 1 号楼 C603

北京图灵文化发展有限公司 图灵读者俱乐部

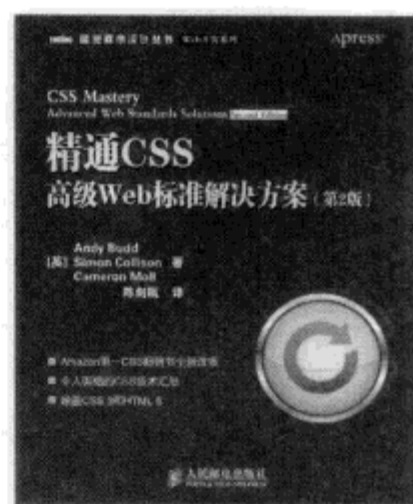




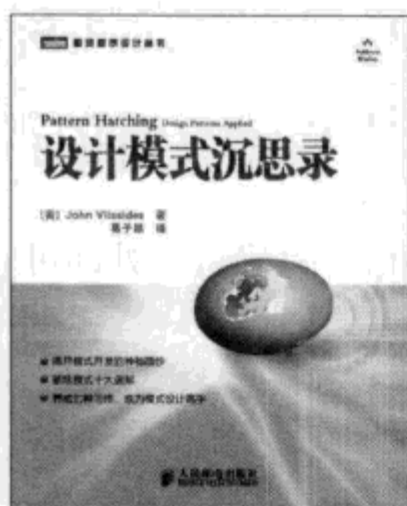
- 书名：结网：互联网产品经理改变世界
书号：978-7-115-22417-0
- 腾讯CEO马化腾作序推荐
 - 作者8年互联网产品经验的总结和提炼
 - 真实案例，一线实战经验
 - 关键概念与实际案例紧密结合



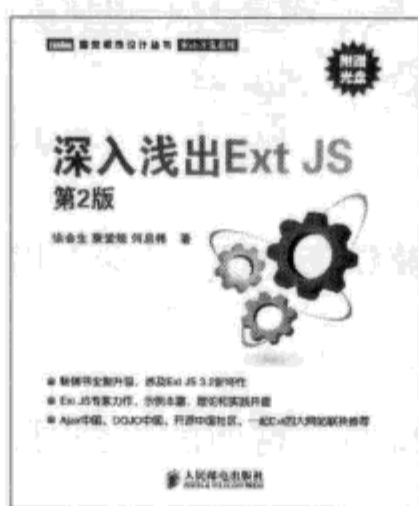
- 书名：博客秘诀：超人气博客是怎样炼成的
书号：978-7-115-21952-7
- 博客专家的金玉良言
 - 提升博客人气不可不用的绝招
 - 亚马逊畅销书



- 书名：精通CSS：高级Web标准解决方案（第2版）
书号：978-7-115-22673-0
- Amazon第一CSS畅销书全新改版
 - 令人叫绝的CSS技术汇总
 - 涵盖CSS和HTML 5



- 书名：设计模式沉思录
书号：978-7-115-22463-7
- 揭开模式开发的神秘面纱
 - 破除模式十大误解
 - 养成七种习惯，成为模式设计高手



- 书名：深入浅出Ext JS(第2版)
书号：978-7-115-22637-2
- 畅销书全新升级，涉及Ext JS 3.2新特性
 - Ext JS专家力作，示例丰富，理论和实践并重
 - Ajax中国、DOJO中国、开源中国社区、一起Ext四大网站联袂推荐



- 书名：.NET设计规范：约定、惯用法与模式（第2版）
书号：978-7-115-22651-8
- 微软.NET Framework设计组的智慧结晶
 - 洞悉.NET技术内幕
 - .NET开发者的必备图书